Nonlinear programming based algorithms to cutting and packing problems(Dissertation_全文)

AUTHOR(S):

Imamichi, Takashi

Nonlinear Programming Based Algorithms to Cutting and Packing Problems

Takashi Imamichi

Department of Applied Mathematics and Physics
Graduate School of Informatics
Kyoto University
Kyoto 606-8501, Japan

March, 2009

Doctoral Dissertation
submitted to Graduate School of Informatics, Kyoto University
in partial fulfillment of the requirement for the degree of
DOCTOR OF INFORMATICS
(Applied Mathematics and Physics)

Preface

Problems of cutting and packing objects in a one, two or three-dimensional space have been extensively studied for a long time because they include numerous real world applications. One-dimensional cutting and packing problem and rectangular cutting and packing problems, which arise from the applications in the production industry such as steel and paper industry, have been well considered so far because finding one-dimensional solutions are relatively easy to solve compared with the cutting and packing problems with complex shapes. However, for the two- or three-dimensional case, it is already a hard problem to just check whether two non-convex polygons overlap or not.

Since both computer technology and computer science have progressed drastically, it has been possible to treat problems with complicated shapes in a practical time. For example, many efficient methods have been obtained recently for cutting and packing problems with complicated shapes such as the two-dimensional irregular strip packing problem. Moreover, cutting and packing problems have applications in other fields than the production industry. For example, restoration of broken objects plays an important role in history science. Also, graph drawing and visualization have an interesting relationship with cutting and packing problems. To visualize information in a two or three-dimensional space, it is often required to show data as a collection of objects, which may have non-convex shape, on a screen in an effective manner, and placing objects with no overlap is one of the basic measures of the effectiveness.

In this thesis, we propose algorithms for two- and three-dimensional cutting and packing problems with objects in complex shapes. In the algorithms, we use local search algorithms that remove overlaps of objects in given layouts by moving them locally, and such a movement of the objects is obtained as a solution to a nonlinear programming. As an example of applications of the local search algorithms, we consider a problem of removing overlaps in label layouts, which arises from a graph drawing problem.

The thesis is organized as follows. We first consider the two-dimensional irregular strip packing problem, where each polygon is allowed to rotate by fixed degrees. We allow penetrations between polygons and protrusions of polygons from a container by penalizing them,

and we formulate a *polygon separation problem* as an unconstrained nonlinear programming problem with a differentiable objective function. We design a *separation algorithm* by applying a nonlinear programming method to the problem and develop an iterated local search algorithm combining the separation algorithm with a swapping operation of two polygons.

Next, we consider a problem that requires to find a layout of objects in a container with no overlap of objects and no protrusion of objects from the container, where the shapes of the container is not necessarily a rectangle and the objects may be three-dimensional. For this problem, we propose a *multi-sphere scheme* which first approximates objects by sphere sets and finds a layout of the approximated objects. As one of the building blocks of the scheme, we design a local search algorithm in a similar way to that for the two-dimensional irregular strip packing problem. Compared with the local search algorithm for the two-dimensional irregular strip packing problem, that for the multi-sphere scheme can handle various motions of objects such as translations in fixed directions and rotations by arbitrary angles. We also design a fast algorithm for detecting collision among spheres as another building block of the multi-sphere scheme. We apply the multi-sphere scheme to three different problems in removing overlaps in label layouts and show experimental results.

We believe that our algorithms are useful in practical applications and the developed techniques provide a new insight into cutting and packing problems. We hope that the works in this thesis will be helpful to advance the study in these topics.

March, 2009

Takashi Imamichi

Acknowledgement

Without support and encouragement from numerous people, I could have never completed this work. I would like to acknowledge them here.

First of all, I would like to express my sincere appreciation to Professor Hiroshi Nagamochi of Kyoto University for his guidance. He gave me much opportunity and continuous support for studying this area. His advice provided insights into research activity for me.

I am deeply grateful to Professor Toshihide Ibaraki of Kwansei Gakuin University. He welcomed me to his laboratory and supervised my bachelor's thesis. He let me notice fascinating aspect of this field.

I would like to express my sincere appreciation to Professor Mutsunori Yagiura of Nagoya University. His hearty and earnest guidance has encouraged me all the time. Several enthusiastic discussions with him were quite exciting and invaluable experience for me. I am indebted to Professor Shinji Imahori of the University of Tokyo, Professor Shunji Umetani of Osaka University, Professor Seok-Hee Hong of the University of Sydney, and Dr. Hideki Hashimoto of Nagoya University for number of helpful comments and suggestions.

I wish to express my gratitude to Professor Koji Nonobe of Hosei University, Professor Liang Zhao of Kyoto University, Professor Takuro Fukunaga of Kyoto University, and all members in Discrete Mathematics laboratory of Kyoto University for their warm friendship in the period I studied at this laboratory.

I am also thankful to Professor Masao Fukushima of Kyoto University and Professor Yoshito Ohta of Kyoto University for serving on my dissertation committee.

Finally, I would like to express my gratitude to my family for their heartfelt cooperation and encouragement.

Contents

1 Introduction **1**

1.1 Optimization Problem 1

1.2 Combinatorial Optimization Problem 2

1.3 Cutting and Packing Problems 4

1.3.1 Two-dimensional Rectangle Packing Problems 5

1.3.2 Two-dimensional Irregular Strip Packing Problem 7

1.3.3 Three-dimensional Problem 9

1.3.4 Sphere Packing Problem 10

1.3.5 Graph Drawing 10

1.4 Organization of this Thesis 11

2 Two-dimensional Irregular Strip Packing Problem **13**

2.1 Introduction 13

2.2 Formulation and Approach 14

2.2.1 Overlap Minimization Problem 15

2.2.2 Entire Algorithm for the Irregular Strip Packing Problem 17

2.3 Computation of Overlap 19

2.3.1 No-fit Polygon 19

2.3.2 Penetration Depth 20

2.3.3 Amount of Overlap 21

2.3.4 Computing NFPs for ILSQN 24

2.4 Iterated Local Search for the Overlap Minimization Problem 25

2.5 Separation Algorithm 25

2.6 Swapping Two Polygons 26

2.6.1 Moving a Polygon 26

2.6.2 Simplifying FindBestPosition 28

2.6.3 Swapping Two Polygons 28

2.6.4 Initial Solution of ILSQN . . . 29
2.7 Experimental Results . . . 31
2.7.1 Environment . . . 31
2.7.2 Parameters . . . 32
2.7.3 Results . . . 34
2.8 Summary . . . 40

3 Multi-sphere Scheme **41**
3.1 Introduction . . . 41
3.2 Algorithms for Multi-sphere Scheme . . . 44
3.2.1 Penalized Rigid Sphere Set Packing Problem . . . 45
3.2.2 Local Search Algorithm . . . 49
3.2.3 Iterated Local Search Algorithm . . . 50
3.3 Collision Detection of Spheres . . . 51
3.3.1 Motivation . . . 51
3.3.2 Related Work . . . 51
3.3.3 Algorithm . . . 52
3.3.4 Slab Partitioning . . . 52
3.3.5 Plane Sweep Method . . . 55
3.3.6 Performance Analysis . . . 56
3.3.7 Uniform Spatial Subdivision . . . 61
3.3.8 Summary . . . 61
3.4 Implementation of the Collision Detection Algorithm . . . 62
3.4.1 Introduction . . . 62
3.4.2 Axis-aligned Bounding Box Based Method . . . 62
3.4.3 New Algorithm . . . 62
3.4.4 Environment . . . 64
3.4.5 Experiments of Collision Detection . . . 64
3.4.6 Experiments of Iterated Local Search . . . 70
3.4.7 Summary . . . 73
3.5 Removing Overlaps in Label Layouts . . . 74
3.5.1 Introduction . . . 74
3.5.2 Problem Definition . . . 75
3.5.3 Experimental Results . . . 76
3.5.4 Summary . . . 89

4 Conclusion **91**

A Unconstrained Nonlinear Programming **97**

A.1 Line Search Methods 98

A.2 Search Directions 99

List of Figures

1.1 (a) An example of the non-guillotine packing; (b) An example of the guillotine packing 6

2.1 An example of a feasible layout of six polygons in container $C(W, L)$ (O is the origin). 15
2.2 Two layers of algorithm ILSQN 17
2.3 Three ways of shrinking the container 18
2.4 An illustration of $\mathrm{NFP}(P_i, P_j)$ (O is the origin) 20
2.5 An illustration of $\mathrm{NFP}(\overline{C}, P_i)$ 20
2.6 The no-fit polygon $\mathrm{NFP}(P_i \oplus \boldsymbol{x}_i, P_j \oplus \boldsymbol{x}_j)$ and the penetration depth $\delta(P_i \oplus \boldsymbol{x}_i, P_j \oplus \boldsymbol{x}_j)$ 21
2.7 The computation of $f_{ij}(\boldsymbol{x})$ and $\nabla f_{ij}(\boldsymbol{x})$ 22
2.8 The medial axis of an NFP and the nearest points on the boundary from inner points $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ 23
2.9 The computation of $g_i(\boldsymbol{x})$ and $\nabla g_i(\boldsymbol{x})$ 24
2.10 The values of $f_{ij}(\boldsymbol{x})$ on arrow A 24
2.11 (a) The original layout; (b) $\partial\,\mathrm{NFP}(P_i(o_i) \oplus \boldsymbol{x}_i, P_1(o))$ $(i = 2, 3)$ and $\partial\,\mathrm{NFP}(\overline{C}, P_1(o))$ 27
2.12 The average efficiencies against π_{side} (left: $S_{\mathrm{init}} = \texttt{sort}$, right: $S_{\mathrm{init}} = \texttt{random}$) 33
2.13 The average efficiencies against $(r_{\mathrm{dec}}, r_{\mathrm{inc}})$ in % (left: $S_{\mathrm{init}} = \texttt{sort}$, right: $S_{\mathrm{init}} = \texttt{random}$) 34
2.14 The average efficiencies against N_{mo} (left: $S_{\mathrm{init}} = \texttt{sort}$, right: $S_{\mathrm{init}} = \texttt{random}$) 34
2.15 The average efficiencies against σ (left: $S_{\mathrm{init}} = \texttt{sort}$, right: $S_{\mathrm{init}} = \texttt{random}$) . 35
2.16 The best solutions for JAKOBS1, JAKOBS2, ALBANO, FU, SHAPES0, SHAPES1, MAO and SHIRTS obtained by ILSQN 38
2.17 The best solutions for SWIM, MARQUES, DIGHE1, DIGHE2, SHAPES2, DAGLI and TROUSERS obtained by ILSQN 39

3.1 The structure of the multi-sphere scheme 42

3.2 Two objects of a sample instance 48
3.3 A layout of the two objects in Figure 3.2 48
3.4 Examples of slab partitioning for 2D and 3D. Grey spheres are assigned to slab B_2 in 2D. Arrows represent the direction in which the plane sweep method sweeps. 54
3.5 An illustration of plane sweep method; To check a collision of $S_i = S_4$ with S_j, the method picks up S_5 and S_6. 56
3.6 The minimal axis-aligned bounding box of $N_A(S)$ for 2D should be totally contained in the bounding box drawn in the bold lines. The arrow represents the direction in which PlaneSweep sweeps spheres. 60
3.7 Examples of randomly generated layouts. There are 10 objects and each object consists of 1000 spheres in each layout. 65
3.8 The average computation time of collision detection with different numbers of objects (the number of spheres per object is 1000). 68
3.9 The average computation time of collision detection with different numbers of spheres per object (the number of object is 10). 69
3.10 The best layouts for 2D instances. 71
3.11 The best layouts for 3D instances. 72
3.12 The smooth approximation of a rectangle by 33 circles for Problem 1. 77
3.13 The rough approximation of a rectangle by 5 circles for Problem 1. 77
3.14 An initial layout of 260 rectangular labels of the metro map instance of Problem 1. 78
3.15 A final layout of the instance in Figure 3.14 with the smooth approximation. 79
3.16 A final layout of the instance in Figure 3.14 with the rough approximation. . 79
3.17 An initial layout of a road map with name labels of intersections. 80
3.18 A final layout of the instance in Figure 3.17. 81
3.19 An example of a road map layout with 112 labels ($\ell_{\text{label}} = 100, \ell_{\text{grid}} = 200$) : (a) an initial layout, (b) a final layout. 83
3.20 An example of a road map layout with 147 labels ($\ell_{\text{label}} = 100, \ell_{\text{grid}} = 150$) : (a) an initial layout, (b) a final layout. 83
3.21 An example of a road map layout with 222 labels ($\ell_{\text{label}} = 100, \ell_{\text{grid}} = 100$) : (a) an initial layout, (b) a final layout. 84
3.22 The approximation of a rectangle in road map instances. 84
3.23 The average computation time for $\ell_{\text{label}} = 200$. 85
3.24 The average ratio of penalty decrease for $\ell_{\text{label}} = 200$. 86

3.25 An example instance with 50 labels of Problem 3: (a) an initial layout, (b) a final layout. 87
3.26 An example instance with 100 labels of Problem 3: (a) an initial layout, (b) a final layout. 88
3.27 An example instance with 200 labels of Problem 3: (a) an initial layout, (b) a final layout. 88
3.28 Magnified figures of Figure 3.26: (a) an initial layout, (b) a final layout. . . . 89
3.29 The average computation time for instances of Problem 3. 89

4.1 Approximating an object by filling spheres. 93
4.2 A joint of two objects. 94
4.3 Another type of a joint of two objects. 95
4.4 A cushion to prevent the blue object moving against the red object. 96

List of Tables

2.1 The information of the benchmark instances for the two-dimensional irregular strip packing problem (cited from [45]) 31
2.2 The length of ILSQN and the efficiency in % of the four algorithms 36
2.3 The computation time in seconds of the four algorithms 37

3.1 The average number of spheres and collisions with different numbers of objects (the number of spheres per object is 1000). 66
3.2 The average number of spheres and collisions with different numbers of spheres per object (the number of objects is 10). 67
3.3 The information of instances. 67
3.4 The total computation time of collision detection in seconds. 70
3.5 The results of ILS_Rigid. 73
3.6 Details of the selected results of road map instances. 86

Chapter 1

Introduction

1.1 Optimization Problem

An *optimization problem* can be intuitively described as a problem that specifies a collection of solutions and asks to find the best one among them. More precisely, an optimization problem consists of a *feasible region* and an *objective function*. A feasible region is a set of all solutions and a solution in the feasible region is called *feasible*. An optimization problem asks to find a solution with the *optimal* objective function value over all feasible solutions. Usually the "optimal" value of an objective function is defined by the *minimum* or *maximum*. A minimization (resp., maximization) problem requires to minimize (resp., maximize) an objective function. In this thesis, we mainly consider minimization problems. A minimization problem is formally described by

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & x \in X, \end{aligned}$$

where f is a given objective function, $\mathbb{R}$ is the set of real numbers, $\mathbb{R}^n$ is the n-dimensional vector space over $\mathbb{R}$, n-dimensional real vector $x \in \mathbb{R}^n$ is a decision variable, and $X \subseteq \mathbb{R}^n$ is a given feasible region. A solution x^* is *optimal* to the optimization problem if and only if $f(x^*) \leq f(x)$ for all $x \in X$. A solution x' is *locally optimal* to the optimization problem if and only if $f(x^*) \leq f(x)$ for all solutions x in the neighbourhood $N(x')$ of x', where a neighbourhood $N(x) \subset X$ of x is defined as a set of solutions that can be obtained from x by a small modification, e.g., $N(x) = \{y \mid \|x - y\| \leq \delta\}$ for a parameter $\delta > 0$.

If the feasible region X is equal to $\mathbb{R}^n$, then the optimization problem is called an *unconstrained optimization problem*; otherwise, it is called a *constrained optimization problem*. As important constrained optimizations, a *linear programming*, *quadratic programming*, and *semidefinite programming* have been extensively studied, for example.

There are many algorithms proposed for the unconstrained optimization. In the case where the gradient of the objective function is available, we can find a locally optimal solution effectively by applying *quasi-Newton method*, *conjugate gradient method* and *limited memory BFGS method*. If both the gradient and the Hessian of the objective function are available, then we can apply *Newton's method*. See Appendix A for more details of unconstrained optimization with gradients. For problems whose gradient and Hessian are not available, Nelder-Mead method and Powell method are proposed, for example. See Nocedal and Wright [80], and Fletcher [36] for textbooks.

1.2 Combinatorial Optimization Problem

If a problem has a *discrete* feasible region such as a set of integer vectors, then it is called a *combinatorial optimization problem*. A variety of combinatorial optimization problems appear in many application fields and numerous algorithms have been proposed so far. In computational complexity theory, an algorithm for a problem is regarded as *efficient* if its time complexity of an algorithm is bounded above by a polynomial of the instance size of every problem instance. We give two example of polynomial solvable problems: *shortest path problem* and *minimum spanning tree problem*.

Shortest path problem

Input: an undirected graph $G = (V, E)$, a non-negative edge length $w : E \to \mathbb{R}_+$, a source $s \in V$, and a sink $t \in V$

Output: a path from s to t with the minimum total edge length

It can be solved by the well-known Dijkstra's algorithm in $O(|E| \log |V|)$ time.

Minimum spanning tree problem

Input: an undirected graph $G = (V, E)$, an edge length $w : E \to \mathbb{R}$

Output: a spanning tree of G with the minimum total edge length

It can be solved by Prim's algorithm in $O(|E| \log |V|)$ time.

There are some problems to which no efficient algorithms are not known. A decision problem asks to answer "yes" or "no" while an optimization problem asks to answer the optimal value. A class *NP* is the set of decision problems such that any yes instance has a certificate that can be verified in polynomial time. A class *NP-complete* is a subclass of NP, which has a property that any problem in NP can be reduced to problems in NP-complete in polynomial time. This means that if some problem in NP-complete can be solved

in polynomial time, then all problems in NP can be solved in polynomial time as well. A class *NP-hard* is a set of optimization problems that is at least as hard as NP-complete. It is strongly believed that *NP-hard* problems admit no polynomial time algorithm. In other words, solving these problems exactly may necessitate enumerating an essential portion of all the solution candidates in a given instance, whose number increases exponentially as the problem size grows. See Garey and Johnson [39] for more details and related topics.

Since the NP-hardness is based on the worst case complexity, it may be possible to solve NP-hard problems efficiently in a practical sense. Representative methods frequently applied to this end are branch-and-bound and dynamic programming, which are algorithm design techniques to find exact optimal solutions by enumerating only promising solutions efficiently [56]. With intensive studies on these exact algorithms and the rapid progress of computer technology, the problem size that can be exactly solved practically has been increasing for some fundamental NP-hard problems with simple structures. However, it has still a large gap from the problem size required to accommodate problems with complex structures that arise in real applications.

In practice, exact solutions are often not required, and good solutions are acceptable if they can be obtained in a reasonable computational time. From this viewpoint, developing approximate or heuristic algorithms is an important research issue.

An approximation algorithm is a theoretical approach that finds in polynomial time a solution providing a theoretical guarantee as an upper bound on how worse the solution is compared to the optimal solution. On the other hand, metaheuristics is a practical approach whose purpose is to find a good solution in a practical time without any theoretical guarantee of the quality of the solution. The performance of metaheuristics is evaluated through computational experiments.

The representative techniques used in approximation algorithms and metaheuristics are greedy method and local search [1]. Greedy method constructs approximate solutions by assigning the values to variables step by step on the basis of local information. Local search behaves as follows. We define neighborhood $N(x)$ of a solution x and choose an initial solution x_0. Local search maintains an *incumbent solution* x^*, which is the best solution obtained so far during a search. It iteratively seeks for a better solution than x^* among the neighborhood of x^*, which is called *neighborhood search*. If it finds one such solution x', it updates the incumbent solution x^* by x'. Otherwise, local search terminates. To design a good local search algorithm, we need to define appropriate neighborhood and a strategy for executing neighborhood search. If neighborhood is narrow, then local search can check all solutions in the neighbourhood quickly, but it is likely to fall into a locally optimal solution due to

the narrow neighbourhood. On the other hand, if neighbourhood is broad, then local search may find a better solution than that by local search with narrow neighbourhood, but the computation is usually expensive. We can observe a similar trade-off in the strategies of neighborhood search, e.g., a strategy is to choose as x' the first solution that is better than the incumbent solution, and another is to choose as x' the best solution over all solutions in the neighbourhood. Note that local search is usually a terminology for combinatorial optimization. Nonlinear programming methods can be regarded as local search in a sense that they iteratively find better solutions in subspaces of the entire solution space, e.g., the line search method and the trust region method.

Metaheuristics is a category of schemes that finds a good solution fast. Since it concentrates on quality of solutions in practical cases, it usually has no theoretical guarantee to the quality of solutions. Generally, metaheuristic algorithms are constructed based on local search algorithms and designed to search for solutions globally by controlling execution of local search. Thus, metaheuristics does not usually depend on specific structures of problems and can be applied to problems from many fields. Various methods are proposed as metaheuristics, e.g., Iterated Local Search, Guided Local Search, Tabu Search, Genetic Algorithm, and Simulated Annealing. See Glover and Kochenberger [42] and Blum et al. [21] for detail of metaheuristics.

1.3 Cutting and Packing Problems

A generic form of *cutting and packing problems* is given as follows.

Cutting and packing problems

Input: a set of objects called *containers* and a set of objects called *items*.

Output: an optimal layout/assignment of the items into the containers.

Constraints:

- no two items do not overlap each other.
- the items are entirely within the containers.

The *optimality* of each cutting and packing problem varies depending on the objective function. Typical objectives are

- to minimize the number of containers.
- to minimize the size of the container.

- to maximize the total profit of the assigned items.

Cutting and packing problems include numerous problems from real world applications and have been extensively studied for a long time from both theoretical and practical point of view. See typologies by Dyckhoff [31] and Wäscher et al. [94], and a review by Hopper and Turton [53] for more on cutting and packing problems.

One-dimensional cutting and packing problems include *Bin packing problem* and *Knapsack problem*. They are known to be NP-hard, and a higher dimensional version of cutting and packing problems is usually obtained by generalizing one-dimensional problems. Hence, most cutting and packing problems are NP-hard.

Knapsack problem

Input: a set of items (each item has a size and a profit), a capacity c of a knapsack.

Output: a feasible subset of items that maximizes the total profit of items, where a subset of items is called feasible if the total size of items does not exceed c.

Bin packing problem

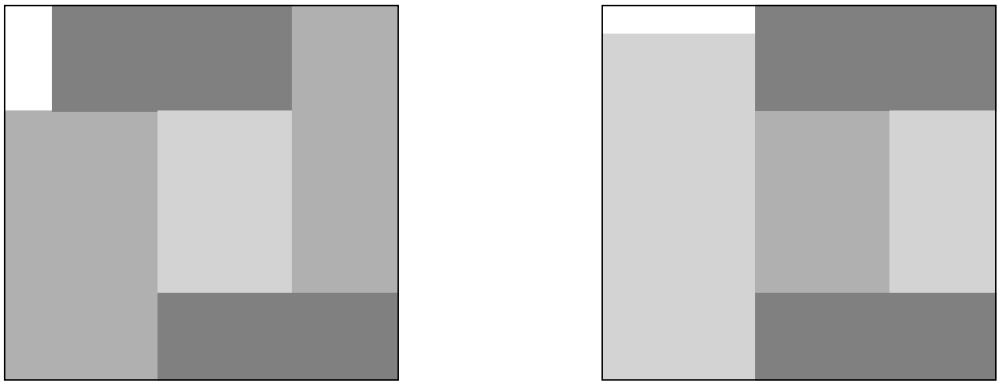
Input: a set of items (each item has a size) and a capacity b of a bin.

Output: a partition of the set of items into the minimum number of disjoint subsets such that the total size of items in each subset does not exceed b.

For two-dimensional cutting and packing problems, given items may have different shapes, e.g., rectangles, convex polygons, circles, non-convex polygons, etc. *Strip packing problem* has been intensively studied as well as bin packing problem and knapsack problem for two-dimensional cutting and packing problems. Strip packing problem, given a rectangular container with a fixed width and a flexible length, asks to place all items in the container so as to minimize the length, which is categorized as *open dimension problem* in a recent typology by Wäscher et al. [94]. Strip packing problem for non-convex polygons is called *irregular strip packing problem*. We review the previous works on two-dimensional strip packing problem, irregular strip packing problem, three-dimensional packing problems, and packing problems with spheres, in Section 1.3.1, Section 1.3.2, Section 1.3.3, and Section 1.3.4, respectively. Section 1.3.5 describes related topics in Graph Drawing.

1.3.1 Two-dimensional Rectangle Packing Problems

Two-dimensional rectangle packing problems have applications in the steel and textile industries, and also has indirect applications to scheduling problems [90] and others [53, 57]. There



(a) (b)

Figure 1.1: (a) An example of the non-guillotine packing; (b) An example of the guillotine packing

are various options on the packing rules; for example, whether rotations of rectangles are allowed or not, whether the width of the container is fixed or not, and so forth.

One of the popular optional restrictions is *guillotine* packing, wherein each cut line must run from end to end on the rectangle being packed [48, 53] (see examples in Figure 1.1). Guillotine packing is important for some manufacturing settings. *Two-stage* packing means that all pieces packed in a rectangle can be obtained by partitioning the rectangle with several horizontal guillotine cuts into strips and then by dividing each of the strips by several vertical guillotine cuts into pieces [49]. In two-stage packing *with trimming*, an additional cutting stage is allowed in order to reduce the length of each piece resulting from the first two stages.

There are also many variations for objective functions. Among those intensively investigated, is *two-dimensional strip packing problem* (2SP) [53], which asks to place all given rectangles without overlap into a container, called a strip, whose width is prescribed, so that the overall length of the container is minimized. A number of heuristic algorithms [15, 24, 52, 57, 58] and exact algorithms [67, 75] have been proposed in the literature. *Perfect packing problem* (PP) is a special case of 2SP that asks to judge whether all given rectangles can be placed, without any overlap or wasted space, into a container with a fixed width and length. Most of the variants of the two-dimensional rectangle packing problem, including 2SP and PP, are known to be NP-hard.

Kenmochi et al. [62] proposed exact algorithms based on the branch-and-bound for 2SP and PP with and without rotations of 90 degrees. Though the size of instances for which exact algorithms work effectively tends to be small, compared to heuristic algorithms, there are many important applications even with a small instances with a few dozens of rectangles. For such small instances, exact algorithms often outperform heuristics, as observed in many

other combinatorial optimization problems. Hifi [48] proposed exact algorithms for 2SP with guillotine constraints without rotations of 90 degrees and solved benchmark instances with up to 22 rectangles. Martello et al. [75] proposed exact algorithms for 2SP without rotations of 90 degrees and succeeded in solving benchmark instances with up to 200 rectangles within a reasonable amount of computation time. Lesh et al. [67] focused on PP without rotations and solved benchmark instances with up to 29 rectangles. Alvarez-Valdes et al. [11] proposed new lower bounds for 2SP and a branch-and-bound algorithm obtained by incorporating a GRASP algorithm and an exact algorithm based on staircase placement.

1.3.2 Two-dimensional Irregular Strip Packing Problem

Two-dimensional irregular strip packing problem requires to place a given set of two-dimensional polygons within a rectangular container so that no polygon overlaps with any other polygon or protrudes from the container, where each polygon is not necessarily convex, We say that such a layout is feasible. The container has a fixed width, while its length can change so that all polygons are placed in it. The objective is to find a feasible layout that minimizes the length of the container. A version of the problem where a finite number of angles are allowed as rotations of polygons has been well studied. The irregular strip packing problem has many applications in material industry such as paper and textile industries, wherein raw materials are usually given in rolls. In textile industry, rotations are usually restricted to 180 degree only because textiles have the grain and may have a drawing pattern. The irregular strip packing problem is known to be NP-hard even without rotation [79].

To our best knowledge, Art [12] first proposed an algorithm for the irregular strip packing problem. The algorithm approximates each of the non-convex polygons by a convex polygon that encloses it and then places them one by one at a feasible bottom left point. Adamowicz and Albano [2] proposed an algorithm that partitions a given set of polygons into several subsets of polygons, then generates for each of the subsets a rectangle enclosure in which the polygons in the subset are placed compactly (i.e., being with a little wasted space), and finally finds a layout of these enclosures. Albano and Sapuppo [10] gave an algorithm that places given polygons one by one at the bottom-left position according to a sequence of the input polygons, where they used tree search to obtain a good sequence. Some approaches for finding a good sequence are based on local search [44, 81].

Mathematical programming was also used for the irregular strip packing problem. *Compaction* and *separation* algorithms based on linear programming have been proposed, e.g., by Li and Milenkovic [70], Bennell and Dowsland [16], and Gomes and Oliveira [45]. Given a feasible layout of given polygons in a container, a *compaction* algorithm translates the poly-

gons in the current layout continuously in order to minimize the length of the container, and it outputs a locally optimal solution. Given an infeasible layout of the given polygons, a *separation* algorithm translates the polygons in the current layout continuously in order to remove the overlap of the polygons.

Bennell and Dowsland [16] combined the bottom-left method and the linear programming based *compaction* algorithm to obtain an algorithm with a better performance. Gomes and Oliveira [45] hybridised the bottom-left heuristics and the linear programming based *compaction* and *separation* algorithms. They further incorporated the method into simulated annealing, and the resulting algorithm updated many best known results on the benchmark instances of the irregular strip packing problem. Burke et al. [23] developed a bottom-left fill heuristic algorithm, and utilized it with hill climbing or tabu search to obtain high quality solutions quickly. Egeblad et al. [32] developed an efficient method that finds the best position of a specified polygon that minimizes its overlap with the current layout by translating the polygon, and they utilized it in guided local search.

Solving jigsaw puzzles is a special case of the irregular strip packing problem. Goldberg et al. [43] and Wolfson et al. [96] proposed algorithms for real jigsaw puzzles and reported that they succeeded to solve jigsaw puzzles with about 200 pieces. Kong and Kimia [66] consider a more general problem, where each piece may be a general non-convex polygon, and proposed an algorithm for the three-dimensional case as well. This generalization has applications for recovering shredded documents or broken pieces of china, and it should provide an important tool in historical science.

Cutting and packing problems with free rotations have not been so intensively studied. Milenkovic [77] proposed a method to make a given layout of polygons compact by translating and rotating them gradually. Recently, Yuping et al. [97] proposed an algorithm to the two-dimensional irregular packing problem with free rotations. Gensane and Ryckelynck [41] proposed an algorithm for a problem that asks to place congruent squares into the minimum square container. Birgin et al. [18] introduced a method of *sentinels* for the decision problem of the overlap of polygons. This method first generates a set of points on each polygon defines that two items overlap or not if and only if any point in the items is contained inside the other item, and formulates a smooth nonlinear decision model that determines whether a set of items overlap or not.

See a review by Hopper and Turton [53] for more on the strip packing problem including the irregular strip packing problem.

1.3.3 Three-dimensional Problem

Packing problems for 3D (three-dimensional) axis aligned boxes have been extensively studied, which are called *container loading problem* in general. There are some variations of problems as in 1D (one-dimensional) and 2D (two-dimensional) problems, e.g., strip packing, knapsack loading and bin packing. Strip packing problem in 3D asks to place all given boxes in a container with fixed width and depth with no overlap so that the required height is minimized. Knapsack loading problem in 3D, where a profit is associated to each box, and asks to place some of all boxes in a container of fixed size so as to maximize the sum of profits of placed boxes. Bin packing problem in 3D requires to minimize the number of containers to place all boxes, where the size of the container is fixed. Pisinger [84] proposed a heuristic algorithm based on *wall-building* approach for the knapsack loading problem. Martello et al. [76] introduced a lower bound on optimal value of the bin packing problem and developed exact and heuristic algorithms based on an extension of the staircase placement for 2D. Bischoff and Marriott [20] conducted experiments to compare 14 heuristics proposed for the strip packing problems in 3D.

Packing boxes is need to measure the capacity of a car trunk. An European standard and an American standard define the capacity of a trunk as the maximum number of the reference boxes that can be placed in the trunk; this problem is called *trunk packing problem*. Compared with the container loading problem, the trunk packing problem treats a container in a more complex shape, and allows non-axis-aligned layouts of boxes. Eisenbrand et al. [33] proposed a heuristic algorithm to the problem, which generates initial solutions by a grid-based combinatorial method and improves them by simulated annealing.

Packing of 3D non-convex objects can be applied to different kinds of applications. Winterfeld [95] dealt with a lapidary cutting problem that asks to obtain the maximum object in a given shape within a larger object, e.g., to obtain a maximum brilliant cut diamond from a diamond gemstone. He formulated this problem as a general semi-infinite program and solved using an interior point method. Ayyadevara et al. [14] studied a problem that asks to stack some identical parts (not necessarily convex in 3D) to minimize a cost function and proposed a heuristic algorithm that divides the non-convex shape of the parts into convex polyhedra and exploits 3D Minkowski sum of the convex polyhedra to find a non-overlap layout. Aladahali et al. [8, 9] considered the *3D component layout problem*, which asks to find a layout of 3D objects with arbitrary shapes so as to achieve the design objectives, and they extended *pattern search method*, which is used for nonlinear optimization without derivatives, to deal with objective functions effectively. See Cagan et al. [25] for a survey of the 3D component layout problem.

1.3.4 Sphere Packing Problem

Numerous types of problems of packing spheres have been also studied so far. The problem of packing circles/spheres has many variations, such as packing circles into a circle [93], packing circles into a rectangle [19], and sphere packing problem in 2D or a higher dimensional space [88]. Wang et al. [93] formulated an unconstrained nonlinear program of the problem, and designed an algorithm by combining a steepest descent method and a procedure for perturbing a layout.

By modeling objects by sphere sets, we can handle more complex shapes in cutting and packing problems. The protein docking, which asks to find the best matching between two molecules in 3D, is an important field of bioinformatics and extensively studied. A molecule is often modeled by a set of spheres. This problem asks not only to find a layout of no two molecules overlap each other but also to minimize scoring functions such as free energies. The *rigid protein docking* is to find a layout of given proteins assuming that proteins do not deform. On the other hand, the *flexible protein docking* allows proteins to deform. Choi and Goyal [28] analyze a combinatorial structure of a rigid docking problem with a simple score function. Choi et al. [27] proposed a local search algorithm for the same rigid docking problem. Besides, various search algorithms are proposed, for example, simulation based methods, Monte Carlo methods, GA based algorithms, etc. See reviews by Halperin et al. [46] and Taylor et al. [89] for more information.

Ferrez [35] proposed a framework for physical simulation of a set of spheres moving in 3D and a fast algorithm to detect the collisions of spheres. He modeled each object by a sphere set and applied the physical simulation and obtained non-overlap layouts of objects efficiently.

Packing spheres is also useful for triangular meshing for the *finite element method* and the *boundary element method*. A technique proposed by Shimada et al. [87] called *bubble mesh* finds a layout of spheres by *bubble packing* a simulation based sphere packing algorithm, and connects the centers of the spheres by constrained Delaunay triangulation and tetrahedrization to form a mesh. In fact, bubble packing has a slightly different feature from the other problems listed above because it allows intersections of spheres to some extent. However, this is one of the interesting applications of the sphere packing.

1.3.5 Graph Drawing

Graph Drawing has been extensively studied over the last twenty years due to its popular application to visualisation in VLSI layout and visualization of computer networks, software engineering, social networks and bioinformatics. As a result, many algorithms and method are available [37]. It has a relationship to cutting and packing problems and we apply our

method to some problems in graph drawing in this thesis. We review the related results in Graph Drawing.

Most algorithms and methods in Graph Drawing deal with abstract graph layout, wherein each node is represented as a point. However, in many real world applications, nodes may have labels with different size and shape. For example, some nodes have long text labels or large images, and they can be represented as boxes or circles as in UML diagrams. Consequently, direct use of layout algorithm for abstract graph often leads to overlapping of nodes (i.e., labels) in the resulting visualization.

In order to visualize graphs with different node sizes, the following three steps are used as a general approach. First, a reasonably good initial layout is created using a graph layout algorithm without considering node size. Second, labels of nodes are added in the layout. Finally, the post processing step to remove node (i.e., label) overlapping is performed.

The problem of removing node overlaps has been well studied for the last ten years by the Graph Drawing community. These can be classified into three different approaches: methods based on force-directed algorithm [38, 47, 54, 69, 73, 78], methods based on the use of Voronoi Diagram [38, 73], and methods using constrained optimization techniques [30, 74].

Furthermore, they differ in their criteria to be optimized. A variation of Force Scan algorithm based on the force-directed method [47, 54, 69, 78] preserves *orthogonal ordering*, i.e., the top-down and left-right relationship between nodes. The problem of transforming a given layout of a graph with overlapping rectangular nodes into a *minimum area* layout that has no node overlapping while preserving the orthogonal order is proved to be NP-complete [47]. The constrained optimization techniques using a quadratic programming approach minimizes the *total change* of node positions while satisfying non-overlap constraints [30, 74].

The time complexity for force-directed methods is $O(n^2)$ [38, 47, 54, 69, 73, 78]. For some special cases, it can be reduced to $O(n \log n)$ [30, 73]. Note that most of the methods find solutions to the problem of overlap removal of *rectangular* labels with *translation* only.

1.4 Organization of this Thesis

As we have seen in this chapter, cutting and packing problems are important because they include various problems that arise from real world applications. However, finding optimal solutions to the problems is considerably difficult in general. Our purpose in this thesis is to propose a general framework to solve two- and three-dimensional cutting and packing problems approximately in a practical time. Moreover, we show that a local search algorithm in the framework itself is useful for some applications.

In Chapter 2, we consider two-dimensional irregular strip packing problem, where each

polygon is allowed to rotate by fixed degrees. In order to check whether a given set of polygons can be placed in a container with fixed size, we formulate a *polygon separation problem* as an unconstrained nonlinear programming problem with a differentiable objective function, where we allow penetrations of polygons and protrusions of polygons from a container by penalizing them. We design a *separation algorithm* by applying nonlinear programming method to the problem and combine it with a swapping operation of two polygons to develop an iterated local search algorithm.

In Chapter 3, we consider a general problem that asks to find a layout of objects in a container with no overlap of objects and no protrusion of objects from the container, where each object is allowed to rotate by any degree. For this problem, we propose a *multi-sphere scheme* as a flexible approach that provides algorithms for different types of packing problems. The scheme first approximates objects by sphere sets and finds a layout of the approximated objects. We then propose a local search algorithm in a similar way to that for the two-dimensional irregular strip packing problem. The local search algorithm in the multi-sphere scheme can handle various kinds of motions such as translations in a fixed directions and rotations by arbitrary angles. We also propose a fast collision detection algorithm, which incorporated into the multi-sphere scheme. We apply the multi-sphere scheme to the label overlap removal as an example of applications and show experimental results.

Chapter 2

Two-dimensional Irregular Strip Packing Problem

2.1 Introduction

The *irregular strip packing problem* is a combinatorial optimization problem that requires to place a given set of two-dimensional polygons within a rectangular container, where each polygon is not necessarily convex, so that no polygon overlaps with other polygons or protrudes from the container. We say that such a layout is *feasible*. The container has a fixed width, while its length can change so that all polygons are placed in it. The objective is to find a feasible layout that minimizes the length of the container. The irregular strip packing problem has a few variations depending on rotations of polygons: (1) rotations of any angle are allowed, (2) a finite number of angles are allowed, (3) no rotation is allowed. Among them, we deal with case (2) in this chapter. Note that case (3) is a special case of (2) in which the number of given orientations for each polygon is one. The irregular strip packing problem has many applications in material industry such as paper and textile industries, where raw materials are usually given in rolls. In textile industry, rotations are usually restricted to 180 degrees because textiles have the grain and may have a drawing pattern. The irregular strip packing problem is known to be NP-hard even without rotation [79].

In this chapter, we propose a new *separation* algorithm based on nonlinear programming. We also give an algorithm that swaps two specified polygons in a layout of polygons so that the overlap in the layout is minimized provided that the positions of the other polygons in a given layout are fixed. We incorporate these algorithms as components in an iterated local search algorithm whose objective is to minimize the total amount of overlap and protrusion of a layout, where a layout may not be completely contained in the container during the algorithm. We then develop an algorithm for the irregular strip packing problem using the iterated local search algorithm, which we call ILSQN because we use quasi-Newton method to solve an

unconstrained optimization problem in the iterated local search algorithm. Computational comparisons on representative benchmark instances disclose that our algorithm is competitive with other existing algorithms. Moreover, our algorithm updates several best known results.

This chapter is organized as follows. We formulate the irregular strip packing problem and illustrate our approach in Section 2.2. We then define functions that measure the amount of overlap and show how to evaluate these functions and their gradients in Section 2.3. We propose an iterated local search algorithm for the overlap minimization problem in Section 2.4 and describe two procedures used in the iterated local search algorithm: a *separation* algorithm based on nonlinear programming and an operation of swapping two polygons in Section 2.5 and Section 2.6, respectively. Finally we show experimental results in Section 2.7 and make a concluding remark in Section 2.8.

2.2 Formulation and Approach

This section gives a mathematical formulation of the irregular strip packing problem and then illustrates an overview of our approach to the problem. For the irregular strip packing problem, we are given a list $\mathcal{P} = (P_1, \ldots, P_n)$ of polygons in a two-dimensional space, a list $\mathcal{O} = O_1 \times \cdots \times O_n$ of the polygons' orientations, where O_i $(1 \leq i \leq n)$ denotes a set of orientations in which P_i can be rotated, and a rectangular container $C = C(W, L)$ with a width $W \geq 0$ and a length L, where W is a constant and L is a nonnegative variable. Polygons in $\mathcal{P}$ may not be convex.

We denote polygon $P_i \in \mathcal{P}$ rotated in degree $o \in O_i$ by $P_i(o)$, which may be written as P_i for simplicity when the orientation is not specified or clear from the context. For convenience, we regard each of polygons $P_i(o)$ $(i = 1, \ldots, n)$ and rectangle C as the set of points inside it including the points on the boundary. For a polygon S, let $\mathrm{int}(S)$ be the interior of S, ∂S be the boundary of S, $\overline{S}$ be the complement $\mathbb{R}^2 \setminus S$ of S, and $\mathrm{cl}(S)$ be the closure $\overline{\mathrm{int}(\overline{S})}$ of S. Translations of polygons can be represented by Minkowski sums as follows. Let $\boldsymbol{x}_i = (x_{i1}, x_{i2})$ $(i = 1, \ldots, n)$ be a translation vector for P_i. Then the polygon obtained by translating polygon P_i by $\boldsymbol{x}_i$ is

$$P_i \oplus \boldsymbol{x}_i = \{\boldsymbol{p} + \boldsymbol{x}_i \mid \boldsymbol{p} \in P_i\}.$$

Recall that $L \geq 0$ is the length of the container C, which is a decision variable to be minimized.

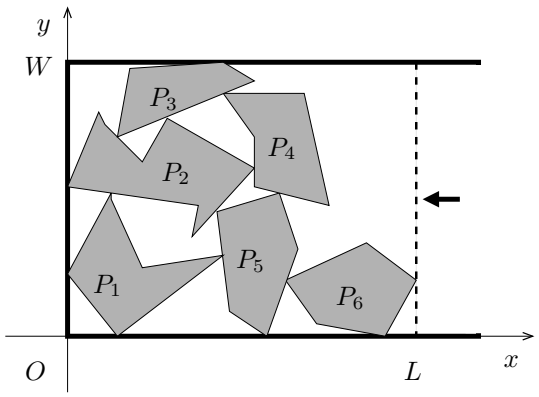


Figure 2.1: An example of a feasible layout of six polygons in container $C(W, L)$ (O is the origin).

The irregular strip packing problem is formally described as follows:

$$\begin{aligned} \text{minimize} \quad & L \\ \text{subject to} \quad & \operatorname{int}(P_i(o_i) \oplus \boldsymbol{x}_i) \cap (P_j(o_j) \oplus \boldsymbol{x}_j) = \emptyset, \quad 1 \le i < j \le n, \\ & (P_i(o_i) \oplus \boldsymbol{x}_i) \subseteq C(W, L), \quad 1 \le i \le n, \\ & L \in \mathbb{R}_+, \\ & o_i \in O_i, \quad 1 \le i \le n, \\ & \boldsymbol{x}_i \in \mathbb{R}^2, \quad 1 \le i \le n. \end{aligned} \tag{2.1}$$

We represent a solution to problem (2.1) with a pair of n-tuples $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ and $\boldsymbol{o} = (o_1, \ldots, o_n)$. Note that a solution $(\boldsymbol{x}, \boldsymbol{o})$ uniquely determines the layout of the polygons. The minimum length L of the container C is formally defined by function

$$\begin{aligned} \mu(\boldsymbol{x}, \boldsymbol{o}) = & \max\{x_1 \mid (x_1, x_2) \in P_i(o_i) \oplus \boldsymbol{x}_i,\ P_i \in \mathcal{P}\} \\ & - \min\{x_1 \mid (x_1, x_2) \in P_i(o_i) \oplus \boldsymbol{x}_i,\ P_i \in \mathcal{P}\}. \end{aligned} \tag{2.2}$$

Figure 2.1 shows an example of a feasible layout of polygons. The length L is decided as described by (2.2).

2.2.1 Overlap Minimization Problem

The problem (2.1) contains three types of variables, L, $\boldsymbol{x}$ and $\boldsymbol{o}$. To construct a building block of an entire algorithm to problem (2.1), we first introduce the *overlap minimization problem*, which requires to find a feasible solution in container C with a fixed length L. For this purpose, we allow polygons to overlap each other and/or protrude from the container during construction of solutions; the amount of overlap and protrusion is penalized in such a way that

any solution with no penalty corresponds to a feasible layout to the current container. More specifically, for a pair $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ and $\boldsymbol{o} = (o_1, \ldots, o_n)$ of lists of translation vectors and orientations of all polygons, let $f_{ij}(\boldsymbol{x}, \boldsymbol{o})$ be a function that measures the amount of overlap of $P_i(o_i)$ and $P_j(o_j)$, and $g_i(\boldsymbol{x}, \boldsymbol{o})$ be a function that measures the amount of protrusion of $P_i(o_i)$ from the container. Then the overlap minimization problem is formulated by

$$\begin{aligned} \text{minimize} \quad & F(\boldsymbol{x}, \boldsymbol{o}) = \sum_{1 \le i < j \le n} f_{ij}(\boldsymbol{x}, \boldsymbol{o}) + \sum_{1 \le i \le n} g_i(\boldsymbol{x}, \boldsymbol{o}) \\ \text{subject to} \quad & \boldsymbol{x} \in \mathbb{R}^{2n}, \quad \boldsymbol{o} \in \mathcal{O}. \end{aligned} \tag{2.3}$$

To solve the overlap minimization problem (2.3), an effective procedure to the problem will be the heart of our algorithm.

Note that the problem (2.3) is an unconstrained nonlinear programming problem. However, we do not attempt to solve this problem by a minimization algorithm such as quasi-Newton method since procedures for evaluating functions f_{ij} and g_i and their gradients would be involved due to variables $\boldsymbol{o}$ for rotations. We treat only variables $\boldsymbol{x}$ for translations while fixing variables for rotations. This enables us to evaluate suitably defined functions f_{ij} and g_i considerably easier by use of an efficient data structure, called *no-fit polygons*, as will be discussed in Section 2.3. Given a solution $(\boldsymbol{x}, \boldsymbol{o})$, we fix the orientation $\boldsymbol{o}$, and introduce the following problem of reducing the total overlap translating polygons, which is called *polygon separation problem*:

$$\begin{aligned} \text{minimize} \quad & F(\boldsymbol{x}) = \sum_{1 \le i < j \le n} f_{ij}(\boldsymbol{x}) + \sum_{1 \le i \le n} g_i(\boldsymbol{x}) \\ \text{subject to} \quad & \boldsymbol{x} \in \mathbb{R}^{2n}, \end{aligned} \tag{2.4}$$

where we omit the indication of $\boldsymbol{o}$ for simplicity. We design a *separation algorithm* to the unconstrained nonlinear programming problem (2.4) by applying quasi-Newton method. The algorithm translates all polygons simultaneously to construct a locally optimal solution.

Since the separation algorithm only translates polygons, we need a procedure for changing the orientations of polygons to handle (2.3). For this, we design a swapping procedure that changes the positions and orientations of two specified polygons to find their best positions and orientations under the condition that the positions and orientations of the other polygons are fixed.

By combining the separation algorithm and the swapping procedure, we construct an iterated local search algorithm, called MinimizeOverlap, to find a solution to the overlap minimization problem (2.3). Given a layout $(\boldsymbol{x}, \boldsymbol{o})$, MinimizeOverlap($\mathcal{P}$, $\mathcal{O}$, $C(W, L)$, $\boldsymbol{x}$, $\boldsymbol{o}$) outputs a new layout $(\boldsymbol{x}^*, \boldsymbol{o}^*)$, which is obtained by modifying $(\boldsymbol{x}, \boldsymbol{o})$, and is a locally optimal solution to the problem (2.3). The details of MinimizeOverlap will be given in Section 2.4.

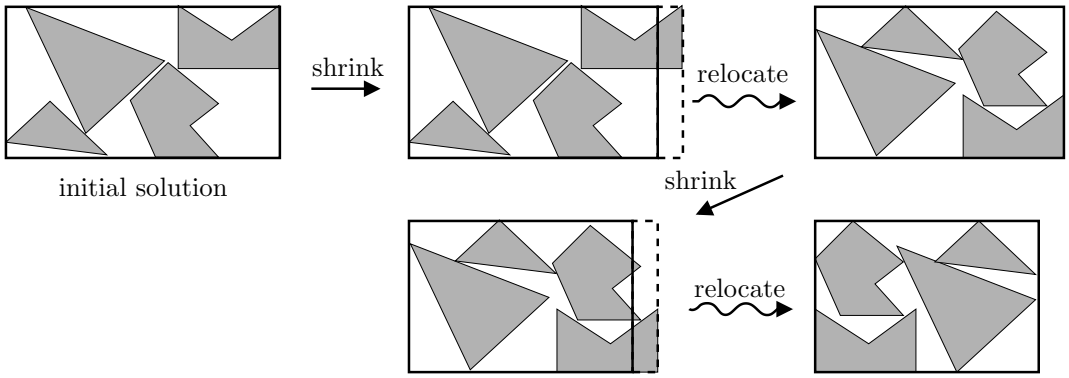


Figure 2.2: Two layers of algorithm ILSQN

2.2.2 Entire Algorithm for the Irregular Strip Packing Problem

In this subsection, we give an entire description of algorithm ILSQN for problem (2.1). ILSQN first generates an initial solution by a method which we will give in Section 2.6.4, and sets the length L of the container so that $C(W, L)$ contains all polygons and the both left and right sides of $C(W, L)$ touch some polygons. Then ILSQN repeats the following two layers of computations until a time limit is reached. One of the two layers is an outer layer that searches the minimum feasible length L^* by shrinking or extending the left and/or right sides of the container. For the current layout $(\boldsymbol{x}_{\rm cur}, \boldsymbol{o}_{\rm cur})$, the outer layer chooses a length L of the container, where $(\boldsymbol{x}_{\rm cur}, \boldsymbol{o}_{\rm cur})$ may be infeasible to the tentatively fixed length L. Then the inner layer, the other layer, improves the current solution $(\boldsymbol{x}_{\rm cur}, \boldsymbol{o}_{\rm cur})$ into a locally optimal solution for the new length L. To find such a solution, the inner layer invokes MinimizeOverlap. Figure 2.2 illustrates the behavior of the algorithm, where "shrink" corresponds to the outer layer and "relocate" corresponds to the inner layer.

We now explain how to execute the outer layer. The execution of the outer layer is described by parameters $r_{\rm dec} \in (0, 1)$, $r_{\rm inc} \in (0, 1)$ and $\pi_{\rm side} \in \{\texttt{left}, \texttt{right}, \texttt{both}\}$. We shrink and extend the length L of the container by factors $r_{\rm dec}$ and $r_{\rm inc}$, respectively. Parameter $\pi_{\rm side}$ determines which side of the container we shrink or extend. To be more precise, when ILSQN changes L to $L - l$, it translates the container to the right by l, 0, and $l/2$ if $\pi_{\rm side} =$ `left`, `right`, and `both`, respectively, as shown in Figure 2.3.

After computing an initial solution $(\boldsymbol{x}, \boldsymbol{o})$, we shorten L by $L := (1 - r_{\rm dec})\mu(\boldsymbol{x}, \boldsymbol{o})$, and execute the inner layer. If the inner layer obtains a feasible layout successfully, then the outer layer shortens the length L of the container by $L := (1 - r_{\rm dec})L$; otherwise, it extends the length by $L := (1 + r_{\rm inc})L$. Algorithm ILSQN is formally described in Algorithm 1, in which we omit indication of $\pi_{\rm side}$ for simplicity.

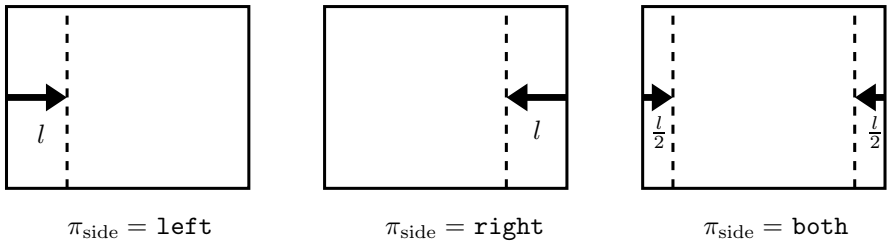


Figure 2.3: Three ways of shrinking the container

Algorithm 1 : ILSQN($\mathcal{P}, \mathcal{O}, W$)

Generate an initial solution $(\boldsymbol{x}, \boldsymbol{o})$; /* see Section 2.6.4 */
$L_{\text{best}} := \mu(\boldsymbol{x}, \boldsymbol{o})$; $(\boldsymbol{x}_{\text{best}}, \boldsymbol{o}_{\text{best}}) := (\boldsymbol{x}, \boldsymbol{o})$;
$L := (1 - r_{\text{dec}})L_{\text{best}}$; $(\boldsymbol{x}_{\text{cur}}, \boldsymbol{o}_{\text{cur}}) := (\boldsymbol{x}, \boldsymbol{o})$;
while within a time limit **do**
  $(\boldsymbol{x}_{\text{cur}}, \boldsymbol{o}_{\text{cur}}) :=$ MinimizeOverlap($\mathcal{P}, \mathcal{O}, C(W, L), \boldsymbol{x}_{\text{cur}}, \boldsymbol{o}_{\text{cur}}$); /* see Algorithm 2 */
  if $(\boldsymbol{x}_{\text{cur}}, \boldsymbol{o}_{\text{cur}})$ is feasible **then**
    $L_{\text{best}} := L$; $(\boldsymbol{x}_{\text{best}}, \boldsymbol{o}_{\text{best}}) := (\boldsymbol{x}_{\text{cur}}, \boldsymbol{o}_{\text{cur}})$;
    $L := (1 - r_{\text{dec}})L_{\text{best}}$
  else
    $L := (1 + r_{\text{inc}})L$;
    if $L \geq L_{\text{best}}$ **then**
      $L := (1 - r_{\text{dec}})L_{\text{best}}$; $(\boldsymbol{x}_{\text{cur}}, \boldsymbol{o}_{\text{cur}}) := (\boldsymbol{x}_{\text{best}}, \boldsymbol{o}_{\text{best}})$
    end if
  end if
end while;
Return $(L_{\text{best}}, \boldsymbol{x}_{\text{best}}, \boldsymbol{o}_{\text{best}})$.

2.3 Computation of Overlap

This section defines suitable functions f_{ij} and g_i in the overlap minimization problem (2.3). Although there are several ways of defining these functions, we use the penetration depth to define them. The gradients of f_{ij} and g_i are important for our separation algorithm in Section 2.5 under the condition that all polygons' orientations are fixed. To compute the gradients of functions f_{ij} and g_i, we use the no-fit polygons. We abbreviate $P_i(o_i)$ as P_i in Section 2.3.1, Section 2.3.2 and Section 2.3.3 for simplicity.

2.3.1 No-fit Polygon

The *no-fit polygon* (NFP) is a data structure that is often used in algorithms for the irregular strip packing problem [2, 10, 16, 44, 45, 81]. Art first applied the same notion, which is called *envelope*, to the irregular strip packing problem [12]. It is also used for other problems such as robotics, in which the no-fit polygon is called configuration-space obstacle, Minkowski difference, etc. Practical algorithms to calculate an NFP of two non-convex polygons have been proposed, e.g., by Bennell et al. [17] and Ramkumar [86].

The no-fit polygon $\mathrm{NFP}(P_i, P_j)$ for an ordered pair of two polygons P_i and P_j is defined by

$$\mathrm{NFP}(P_i, P_j) = \mathrm{int}(P_i) \oplus (-\mathrm{int}(P_j)) = \{\boldsymbol{v} - \boldsymbol{w} \mid \boldsymbol{v} \in \mathrm{int}(P_i),\ \boldsymbol{w} \in \mathrm{int}(P_j)\}.$$

The no-fit polygon has the following important properties:

- $P_i \oplus \boldsymbol{x}_i$ and $P_j \oplus \boldsymbol{x}_j$ overlap if and only if $\boldsymbol{x}_j - \boldsymbol{x}_i \in \mathrm{NFP}(P_i, P_j)$.
- $P_i \oplus \boldsymbol{x}_i$ touches $P_j \oplus \boldsymbol{x}_j$ if and only if $\boldsymbol{x}_j - \boldsymbol{x}_i \in \partial\,\mathrm{NFP}(P_i, P_j)$.
- $P_i \oplus \boldsymbol{x}_i$ and $P_j \oplus \boldsymbol{x}_j$ are separated if and only if $\boldsymbol{x}_j - \boldsymbol{x}_i \notin \mathrm{cl}(\mathrm{NFP}(P_i, P_j))$.

Hence the problem of checking whether two polygons overlap or not becomes an easier problem of checking whether a point is in a polygon or not. When P_i and P_j are both convex, $\partial\,\mathrm{NFP}(P_i, P_j)$ can be computed by the following simple procedure. We first place the reference point of P_i at the origin, and slide P_j around P_i, i.e., translate P_j having it keep touching with P_i. Then the trace of the reference point of P_j is $\partial\,\mathrm{NFP}(P_i, P_j)$. Figure 2.4 shows an example of $\mathrm{NFP}(P_i, P_j)$ of two polygons P_i and P_j.

We can also check whether a polygon P_i protrudes from the container C or not by using

$$\mathrm{NFP}(\overline{C}, P_i) = \mathrm{int}(\overline{C}) \oplus (-\mathrm{int}(P_i)) = \{\boldsymbol{v} - \boldsymbol{w} \mid \boldsymbol{v} \in \mathbb{R}^2 \setminus C,\ \boldsymbol{w} \in \mathrm{int}(P_i)\},$$

which is the complement of a rectangle whose boundary is the trajectory of the reference point of P_i when we slide P_i inside the container C. See an example in Figure 2.5. The following properties are derived from those of the no-fit polygon:

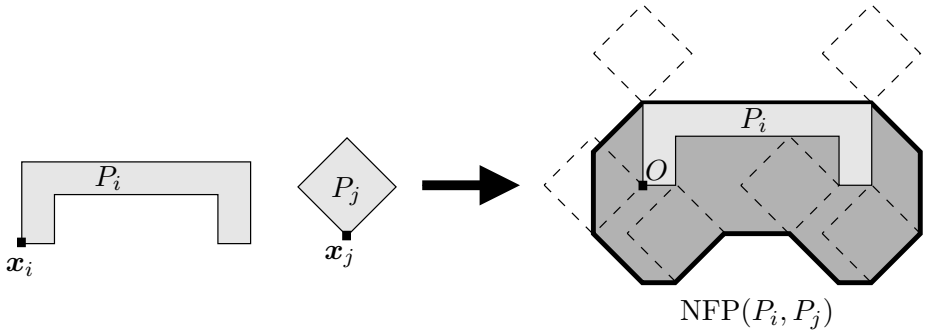


Figure 2.4: An illustration of $\mathrm{NFP}(P_i, P_j)$ (O is the origin)

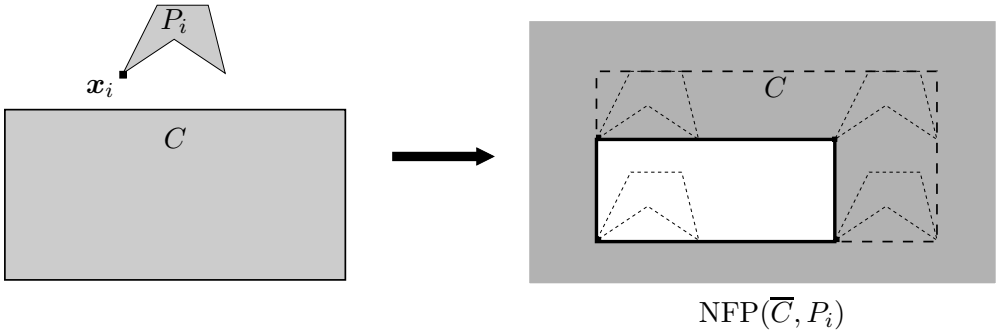


Figure 2.5: An illustration of $\mathrm{NFP}(\overline{C}, P_i)$

- $P_i \oplus \boldsymbol{x}_i$ protrudes from C if and only if $\boldsymbol{x}_i \in \mathrm{NFP}(\overline{C}, P_i)$.
- $P_i \oplus \boldsymbol{x}_i$ is contained in C and touches ∂C if and only if $\boldsymbol{x}_i \in \partial\,\mathrm{NFP}(\overline{C}, P_i)$.
- $P_i \oplus \boldsymbol{x}_i$ is contained in C and does not touch ∂C if and only if $\boldsymbol{x}_i \notin \mathrm{cl}(\mathrm{NFP}(\overline{C}, P_i))$.

To check if a polygon $P_i \oplus \boldsymbol{x}_i$ protrudes from C, Gomes and Oliveira [44, 45] introduced the *inner-fit rectangle*, which is equivalent to $\overline{\mathrm{NFP}(\overline{C}, P_i)}$.

2.3.2 Penetration Depth

The *penetration depth* (also known as the intersection depth) is an important notion used for robotics, computer vision and so on [4, 29, 65]. The penetration depth of two objects S and T in the d-dimensional space is defined by the minimum translational distance to separate them, i.e.,

$$\delta(S, T) = \min\{\|\boldsymbol{x}\| \mid \mathrm{int}(S) \cap (T \oplus \boldsymbol{x}) = \emptyset,\ \boldsymbol{x} \in \mathbb{R}^d\},$$

where $\|\cdot\|$ denotes the Euclidean norm. If two polygons P_i and P_j do not overlap, the penetration depth of them is zero. Otherwise, the penetration depth of two overlapping

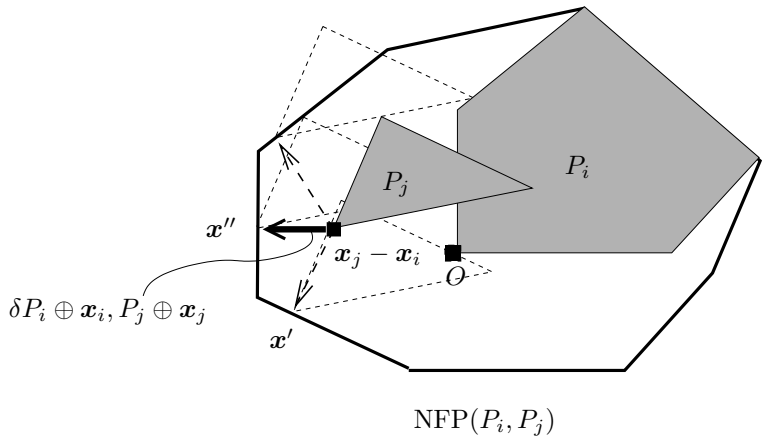


Figure 2.6: The no-fit polygon $\mathrm{NFP}(P_i \oplus \boldsymbol{x}_i, P_j \oplus \boldsymbol{x}_j)$ and the penetration depth $\delta(P_i \oplus \boldsymbol{x}_i, P_j \oplus \boldsymbol{x}_j)$

polygons P_i and P_j is computed by

$$\delta(P_i, P_j) = \min\{\|\boldsymbol{z}\| \mid \mathrm{int}(P_i) \cap (P_j \oplus \boldsymbol{z}) = \emptyset,\ \boldsymbol{z} \in \mathbb{R}^2\},$$

We can separate two polygons P_i and P_j by translating the reference point of P_j to a point $\boldsymbol{x}'$ on $\partial\,\mathrm{NFP}(P_i, P_j)$ (See Figure 2.6). Hence $\delta(P_i \oplus \boldsymbol{x}_i, P_j \oplus \boldsymbol{x}_j)$ is the minimum distance from $\boldsymbol{x}_j - \boldsymbol{x}_i$ to $\partial\,\mathrm{NFP}(P_i, P_j)$. Figure 2.6 shows the relationship between the penetration depth and the NFP. The solid and dashed arrows are examples of translations to the boundary of the NFP and the dotted polygons are the polygons of P_j translated by the vectors represented by these arrows. The solid arrow $\boldsymbol{x}'' - \boldsymbol{x}_j$ has the minimum distance among all translations, giving the penetration depth $\delta(P_i \oplus \boldsymbol{x}_i, P_j \oplus \boldsymbol{x}_j)$.

2.3.3 Amount of Overlap

We define functions f_{ij} and g_i in problem (2.3) using the penetration depth. To represent the amount of overlap between P_i and P_j, we define f_{ij} by

$$f_{ij}(\boldsymbol{x}) = \delta(P_i \oplus \boldsymbol{x}_i, P_j \oplus \boldsymbol{x}_j)^m,\ \ 1 \le i < j \le n,$$

where $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ and m is a positive parameter. Similarly we define $g_i(\boldsymbol{x})$ by

$$g_i(\boldsymbol{x}) = \delta(\mathrm{cl}(\overline{C}), P_i \oplus \boldsymbol{x}_i)^m, \quad 1 \le i \le n.$$

In order to apply efficient algorithms for solving the nonlinear program to the polygon separation problem (2.4), we need to compute the values of $f_{ij}(\boldsymbol{x})$ and $g_i(\boldsymbol{x})$ and their gradients for a given solution $(\boldsymbol{x}, \boldsymbol{o})$, where all polygons' orientations $\boldsymbol{o}$ are fixed. We describe

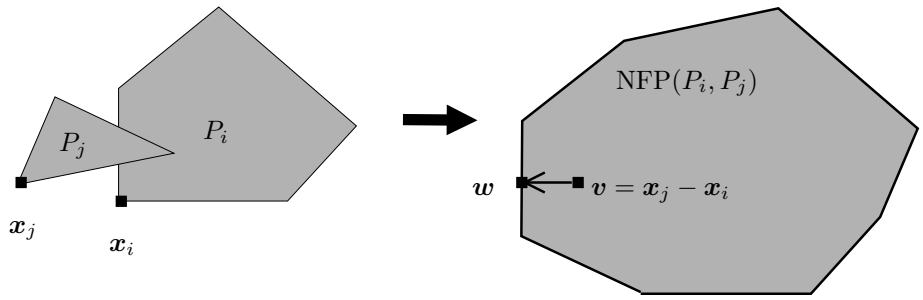


Figure 2.7: The computation of $f_{ij}(\boldsymbol{x})$ and $\nabla f_{ij}(\boldsymbol{x})$

below how we realize such computation. Let $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ be the translation vectors of P_i and P_j, respectively, and denote $\boldsymbol{v} = \boldsymbol{x}_j - \boldsymbol{x}_i$ for convenience. We first consider how to compute $f_{ij}(\boldsymbol{x})$ and $\nabla f_{ij}(\boldsymbol{x})$, and later explain the case of $g_i(\boldsymbol{x})$ and $\nabla g_i(\boldsymbol{x})$. There are three cases for the computation of $f_{ij}(\boldsymbol{x})$ and $\nabla f_{ij}(\boldsymbol{x})$.

Case 1: two polygons P_i and P_j do not overlap. In this case, we see that $f_{ij}(\boldsymbol{x}) = 0$ and $\nabla f_{ij}(\boldsymbol{x}) = \boldsymbol{0}$.

Case 2: two polygons P_i and P_j overlap (i.e., $f_{ij}(\boldsymbol{x}) > 0$) and the nearest point on $\partial\,\mathrm{NFP}(P_i, P_j)$ from $\boldsymbol{v}$ is unique. See an example in Figure 2.7. Let $\boldsymbol{w}$ be the nearest point and let $\boldsymbol{z} = \boldsymbol{w} - \boldsymbol{v}$. Because the variable $\boldsymbol{x}$ is a list of n two-dimensional vectors, $\nabla f_{ij}(\boldsymbol{x})$ is also a list of the same size; hence we denote $\nabla f_{ij}(\boldsymbol{x}) = (\nabla_1 f_{ij}(\boldsymbol{x}), \ldots, \nabla_n f_{ij}(\boldsymbol{x}))$, where $\nabla_k = (\partial/\partial x_{k1}, \partial/\partial x_{k2})$, $1 \le k \le n$. Then, $f_{ij}(\boldsymbol{x})$ and $\nabla f_{ij}(\boldsymbol{x})$ for $1 \le i < j \le n$ are given by

$$\begin{aligned} f_{ij}(\boldsymbol{x}) &= \|\boldsymbol{z}\|^m, \\ \nabla_i f_{ij}(\boldsymbol{x}) &= -\nabla_j f_{ij}(\boldsymbol{x}) = m\|\boldsymbol{z}\|^{m-2}\boldsymbol{z}, \\ \nabla_k f_{ij}(\boldsymbol{x}) &= \boldsymbol{0}, \quad k \in \{1, \ldots, n\} \setminus \{i, j\}. \end{aligned} \tag{2.5}$$

Every $\nabla_k f_{ij}(\boldsymbol{x})$ except $\nabla_i f_{ij}(\boldsymbol{x})$ and $\nabla_j f_{ij}(\boldsymbol{x})$ is zero because only P_i and P_j can contribute to the overlap $f_{ij}(\boldsymbol{x})$.

Case 3: $f_{ij}(\boldsymbol{x}) > 0$ and the nearest point from $\boldsymbol{v}$ to $\partial\,\mathrm{NFP}(P_i, P_j)$ is not unique. In this case, ∇f_{ij} is not differentiable at $\boldsymbol{x}$; however, we choose one of the nearest points arbitrarily as $\boldsymbol{w}$ and calculate $\nabla f_{ij}(\boldsymbol{x})$ with (2.5) as in Case 2.

Case 2 and Case 3 are distinguished in reference to the *medial axis* [5, 26] of $\mathrm{NFP}(P_i, P_j)$. The medial axis of a polygon P is defined by the trace of the centers of all circles contained in P that touch at least two sides of ∂P. Figure 2.8 shows an example of an NFP and its medial axis. The thick solid polygon is an NFP, $s_1, \ldots, s_7$ are the edges of the NFP, and the dashed lines are the medial axis of the NFP. For the two points $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, the arrows indicate the nearest point on the NFP from $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, respectively. The nearest point on the NFP's

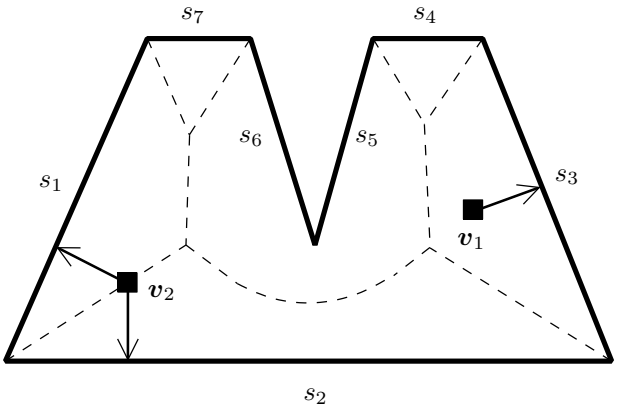


Figure 2.8: The medial axis of an NFP and the nearest points on the boundary from inner points $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$

boundary from a point $\boldsymbol{v}$ in the NFP is unique if and only if $\boldsymbol{v}$ is not on the medial axis of the NFP. In Figure 2.8, $\boldsymbol{v}_1$ is not on the medial axis and it has the unique nearest point on s_3. On the other hand $\boldsymbol{v}_2$ is on the medial axis and it has two nearest points on s_1 and s_2. Note that $\boldsymbol{v}$ can have more than one nearest point only when $\boldsymbol{v}$ is on the medial axis of the NFP. Such a case is rare because the search basically tries to change $\boldsymbol{v}$ so that it moves away from the medial axis in order to minimize the sum of $f_{ij}(\boldsymbol{x})$.

We compute $g_i(\boldsymbol{x})$ and $\nabla g_i(\boldsymbol{x})$ similarly as in the case of $f_{ij}(\boldsymbol{x})$ and $\nabla f_{ij}(\boldsymbol{x})$. If P_i is contained in C, we simply return $g_i(\boldsymbol{x}) = 0$ and $\nabla g_i(\boldsymbol{x}) = \boldsymbol{0}$. We consider a polygon P_i that protrudes from the container C (i.e., $g_i(\boldsymbol{x}) > 0$). See an example in Figure 2.9. Let $\boldsymbol{w}$ be the nearest point on $\partial\,\mathrm{NFP}(\overline{C}, P_i)$ from $\boldsymbol{x}_i$ and $\boldsymbol{z} = \boldsymbol{w} - \boldsymbol{x}_i$; the nearest point is always unique in this case. We denote $\nabla g_i(\boldsymbol{x}) = (\nabla_1 g_i(\boldsymbol{x}), \ldots, \nabla_n g_i(\boldsymbol{x}))$. Then, $g_i(\boldsymbol{x})$ and its gradient for $1 \leq i < j \leq n$ are given by

$$\begin{aligned} g_i(\boldsymbol{x}) &= \|\boldsymbol{z}\|^m, \\ \nabla_i g_i(\boldsymbol{x}) &= -m\|\boldsymbol{z}\|^{m-2}\boldsymbol{z}, \\ \nabla_k g_i(\boldsymbol{x}) &= \boldsymbol{0}, \quad k \in \{1, \ldots, n\} \setminus \{i\}. \end{aligned} \tag{2.6}$$

Every $\nabla_k g_i(\boldsymbol{x})$ except $\nabla_i g_i(\boldsymbol{x})$ is zero because only P_i has influence on its protrusion from the container.

In Case 2, $f_{ij}(\boldsymbol{x})$ is not differentiable at $\boldsymbol{x}$ if and only if P_i and P_j touch each other and $0 < m \leq 1$. Similarly $g_i(\boldsymbol{x})$ is not differentiable at $\boldsymbol{x}$ if and only if P_i is contained in C, touches C and $0 < m \leq 1$. We avoid choosing such m as will be discussed below. We can thus calculate the gradient of the objective function of (2.3).

The positive parameter m determines the differentiability of $f_{ij}(\boldsymbol{x})$ and $g_i(\boldsymbol{x})$. Figure 2.10

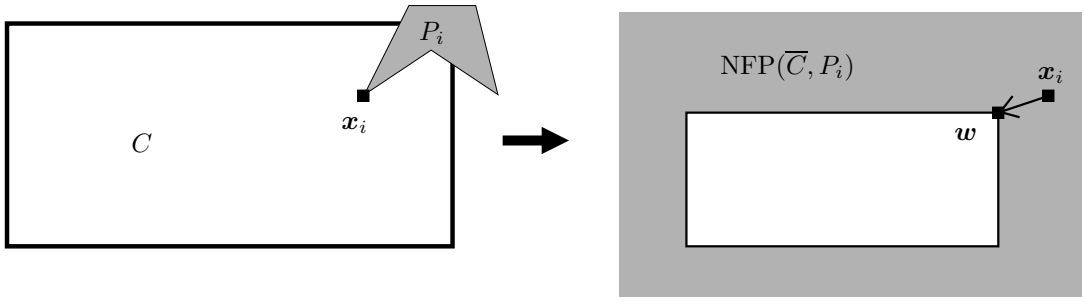


Figure 2.9: The computation of $g_i(\boldsymbol{x})$ and $\nabla g_i(\boldsymbol{x})$

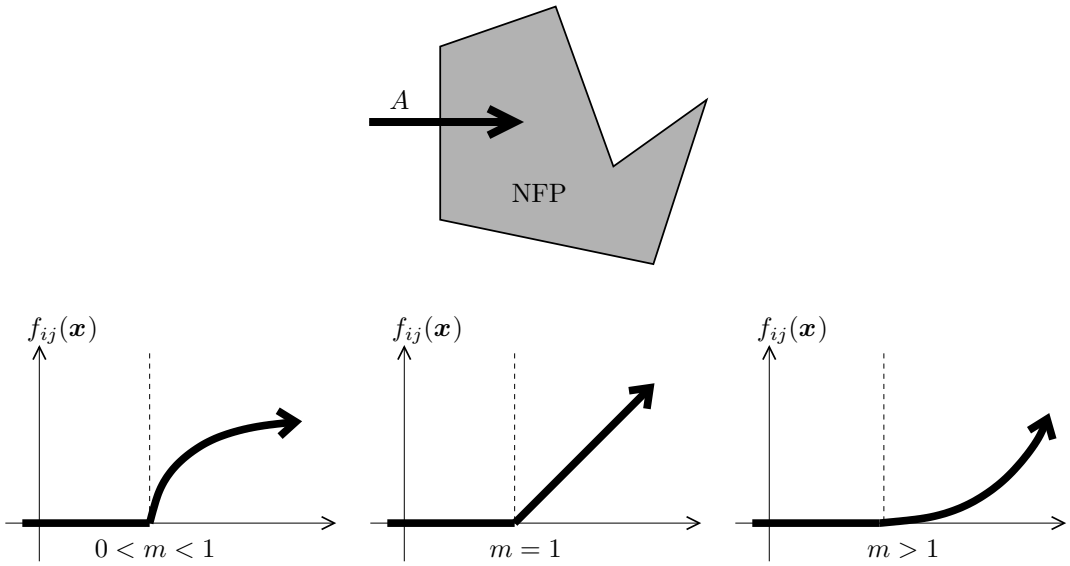


Figure 2.10: The values of $f_{ij}(\boldsymbol{x})$ on arrow A

shows the change of $f_{ij}(\boldsymbol{x})$ along the arrow from the outside to the inside of an NFP. At the boundary of the NFP, $f_{ij}(\boldsymbol{x})$ is nondifferentiable for $0 < m \leq 1$, while it is differentiable for $m > 1$. Moreover, $\nabla f_{ij}(\boldsymbol{x})$ in (2.5) becomes simpler for $m = 2$ because term $\|z\|^{m-2}$ disappears. The situation is the same for $g_i(\boldsymbol{x})$, and hence we let $m = 2$ in our experiments.

2.3.4 Computing NFPs for ILSQN

In the previous subsections, we show how to use no-fit polygons to evaluate functions f_{ij} and g_i and their gradients, where the orientations of polygons are fixed for simplicity. However, in our algorithm ILSQN, we need to use no-fit polygons NFP$(P_i(o_i), P_j(o_j))$ and NFP$(\overline{C}, P_i(o_i))$ for all possible orientations o_i and o_j. We thus compute all NFP$(P_i(o_i), P_j(o_j))$, $o_i \in O_i$, $o_j \in O_j$, $1 \leq i < j \leq n$ beforehand and utilize them in ILSQN.

2.4 Iterated Local Search for the Overlap Minimization Problem

In this section, we formally describe MinimizeOverlap, our iterated local search algorithm for the overlap minimization problem (2.3) introduced in Section 2.2.1. MinimizeOverlap invokes the following algorithms.

- SwapTwoPolygons: an operation of swapping two specified polygons in a given layout considering rotations, where the other polygons and the length L of the container are fixed. (The detail is given in Section 2.6.)
- Separate: an algorithm that translates all polygons in a given layout simultaneously to reduce the total amount of overlap and protrusion, where the length L of the container is fixed. It does not consider rotation. (The detail is given in Section 2.5.)

MinimizeOverlap maintains the earliest solution that minimizes the objective function F of (2.3) among those searched by then as the incumbent solution $(\boldsymbol{x}_{\mathrm{inc}}, \boldsymbol{o}_{\mathrm{inc}})$, which will be used for generating the next initial solution. MinimizeOverlap first perturbs the incumbent solution by swapping two randomly chosen polygons P_i and P_j calling SwapTwoPolygons($\mathcal{P}$, $\mathcal{O}, C$, $\boldsymbol{x}_{\mathrm{inc}}, \boldsymbol{o}_{\mathrm{inc}}$, P_i, P_j), and then translates all polygons simultaneously calling Separate($\mathcal{P}, \mathcal{O}, C, \boldsymbol{x}_{\mathrm{init}}, \boldsymbol{o}_{\mathrm{init}}$) to obtain a locally optimal solution $(\boldsymbol{x}_{\mathrm{lopt}}, \boldsymbol{o}_{\mathrm{lopt}})$ of (2.3). Since Separate does not rotate polygons, $\boldsymbol{o}_{\mathrm{lopt}} = \boldsymbol{o}_{\mathrm{init}}$ holds. If the locally optimal solution has less overlap than the incumbent solution does, we update the incumbent solution with the locally optimal solution. MinimizeOverlap stops these operations if it fails to update the incumbent solution after a prescribed number N_{mo} of consecutive calls to local search. The algorithm is formally described in Algorithm 2, where $(\boldsymbol{x}_{\mathrm{inc}}, \boldsymbol{o}_{\mathrm{inc}})$ is an initial layout given to the algorithm.

2.5 Separation Algorithm

This section describes our separation algorithm Separate, which is used as local search in MinimizeOverlap. The polygon separation problem (2.4) introduced in Section 2.2.1 is an unconstrained nonlinear programming problem, where all polygons' orientations $\boldsymbol{o}$ and the length L of the container are fixed. The objective function F of (2.4) and its gradient ∇F are efficiently computable because ∇f_{ij} and ∇g_i are computable with the no-fit polygons as described in Section 2.3.3. Separate($\mathcal{P}$, $\mathcal{O}$, C, $\boldsymbol{x}$, $\boldsymbol{o}$) is thus realized as follows: Separate first applies quasi-Newton method to the polygon separation problem (2.4) by using the current layout $(\boldsymbol{x}, \boldsymbol{o})$ as an initial solution and Separate returns a locally optimal solution

Algorithm 2 : MINIMIZEOVERLAP($\mathcal{P}, \mathcal{O}, C, \boldsymbol{x}_{\text{inc}}, \boldsymbol{o}_{\text{inc}}$)

```
k := 0;
while k < N_mo do
  Randomly choose P_i and P_j from P;
  (x_init, o_init) := SWAPTWOPOLYGONS(P, O, C, x_inc, o_inc, P_i, P_j);
  (x_lopt, o_lopt) := SEPARATE(P, O, C, x_init, o_init);
  k := k + 1;
  if F(x_lopt, o_lopt) < F(x_inc, o_inc) then
    (x_inc, o_inc) := (x_lopt, o_lopt);
    k := 0
  end if
end while;
Return (x_inc, o_inc).
```

delivered by quasi-Newton method. SEPARATE translates all polygons simultaneously and (usually) slightly to reduce the total amount of overlap and protrusion, where no rotation is considered.

2.6 Swapping Two Polygons

2.6.1 Moving a Polygon

ILSQN perturbs a locally optimal solution by swapping two polygons in the solution. Instead of just exchanging two polygons P_i and P_j in their reference points, we attempt to find their positions with the least overlap. FINDBESTPOSITION($\mathcal{P}$, $\mathcal{O}$, C, $\boldsymbol{x}$, $\boldsymbol{o}$, P_i) is a heuristic algorithm to find a minimum overlap position of a specified polygon P_i without changing the positions of the other polygons, while considering all possible orientations $o \in O_i$ of P_i. Let $\mathcal{N}(o)$ be the set of polygon boundaries $\partial\,\mathrm{NFP}(P_k(o_k) \oplus \boldsymbol{x}_k, P_i(o))$, $P_k \in \mathcal{P} \setminus \{P_i\}$ and $\partial\,\mathrm{NFP}(\overline{C}, P_i(o))$, $\mathcal{V}(o)$ be the set of vertices of $\mathcal{N}(o)$, and $\mathcal{I}(o)$ be the set of edge intersections of $\mathcal{N}(o)$. For each point $\boldsymbol{v} \in \mathcal{V}(o) \cup \mathcal{I}(o)$, the heuristics computes the overlap of $P_i(o) \oplus \boldsymbol{v}$ with the other polygons, and finds the position with the least overlap, where $\boldsymbol{x}$ is a list of the translation vectors of polygons, $\boldsymbol{o}$ is a list of the orientations of polygons, and the amount of overlap is computed by the objective function $F(\boldsymbol{x}, \boldsymbol{o})$ of the overlap minimization problem (2.3). By repeating these operations for all orientations $o \in O_i$ of P_i, FINDBESTPOSITION seeks the best position and orientation. The algorithm is formally described in Algorithm 3, where $(\boldsymbol{x})_i$ denotes the ith element of $\boldsymbol{x}$ and $(\boldsymbol{o})_i$ denotes the ith element of $\boldsymbol{o}$.

Figure 2.11 shows an example in which FINDBESTPOSITION is searching for the best position for a square $P_1(o)$ with a fixed orientation $o \in O_1$ in the left layout. Figure 2.11(b) shows $\mathrm{NFP}(P_2(o_2) \oplus \boldsymbol{x}_2, P_1(o))$, $\mathrm{NFP}(P_3(o_3) \oplus \boldsymbol{x}_3, P_1(o))$, and $\mathrm{NFP}(\overline{C}, P_1(o))$. The circles in

Algorithm 3 : FINDBESTPOSITION($\mathcal{P}, \mathcal{O}, C, \boldsymbol{x}, \boldsymbol{o}, P_i$)

$F^* := +\infty$;
for each $o \in O_i$ **do**
  Compute NFP($P_k(o_k) \oplus \boldsymbol{x}_k, P_i(o)$) ($P_k \in \mathcal{P} \setminus \{P_i\}$) and NFP($\overline{C}, P_i(o)$);
  for each $\boldsymbol{v} \in \mathcal{V}(o) \cup \mathcal{I}(o)$ **do**
    $\boldsymbol{x}' := \boldsymbol{x}$; $(\boldsymbol{x}')_i := \boldsymbol{v}$;
    $\boldsymbol{o}' := \boldsymbol{o}$; $(\boldsymbol{o}')_i := o$;
    if $F(\boldsymbol{x}', \boldsymbol{o}') < F^*$ **then**
      $F^* := F(\boldsymbol{x}', \boldsymbol{o}')$; $\boldsymbol{v}^* := \boldsymbol{v}$; $o^* := o$
    end if
  end for
end for;
Return $(\boldsymbol{v}^*, o^*)$.

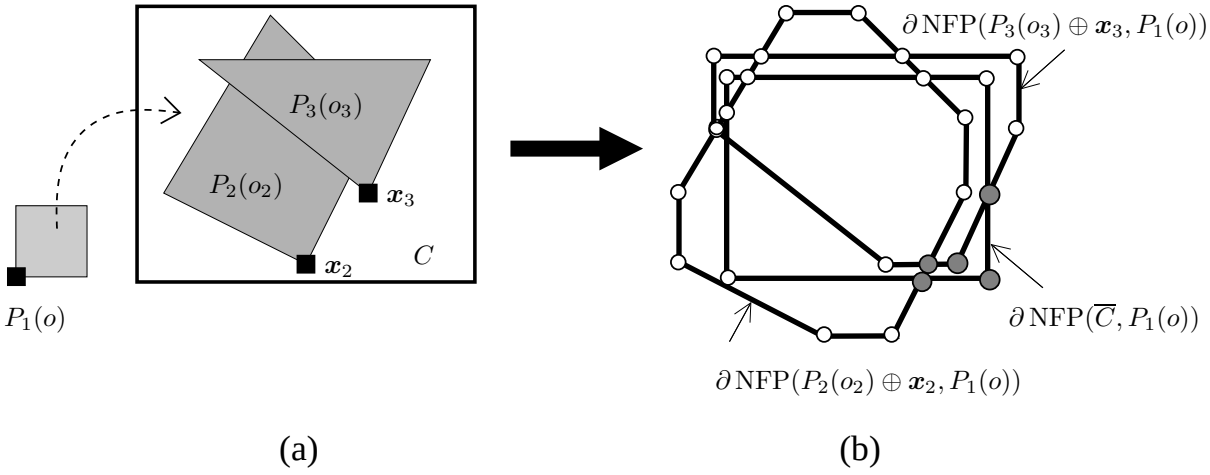


Figure 2.11: (a) The original layout; (b) $\partial\,\mathrm{NFP}(P_i(o_i) \oplus \boldsymbol{x}_i, P_1(o))$ ($i = 2, 3$) and $\partial\,\mathrm{NFP}(\overline{C}, P_1(o))$

Figure 2.11(b) represent $\mathcal{V}(o)$ and $\mathcal{I}(o)$. FINDBESTPOSITION finds a point that corresponds to a layout with the least overlap. Each grey circle in Figure 2.11(b) corresponds to a position that has no overlap in Figure 2.11(a).

FINDBESTPOSITION has an important property described in Lemma 1.

Lemma 1 FINDBESTPOSITION($\mathcal{P}, \mathcal{O}, C, \boldsymbol{x}, \boldsymbol{o}, P_i$) *always finds a point* $\hat{\boldsymbol{v}} \in \mathbb{R}^2$ *and an orientation* $\hat{o} \in O_i$ *of polygon* P_i *such that* $P_i(\hat{o}) \oplus \hat{\boldsymbol{v}}$ *neither overlaps with the other polygons nor protrudes from the container* C *if there exists such a pair of a point and an orientation.*

Proof: Assume that there exists a point $\hat{\boldsymbol{v}} \in \mathbb{R}^2$ and an orientation $\hat{o} \in O_i$ of polygon P_i such that $P_i(\hat{o}) \oplus \hat{\boldsymbol{v}}$ neither overlaps with the other polygons nor protrudes from the container C. Since FINDBESTPOSITION tries all orientations in O_i, it chooses $o = \hat{o}$ in the outer for-loop.

Note that $P_i(o) \oplus \boldsymbol{v}$ does not overlap with $P_k(o_k) \oplus \boldsymbol{x}_k$ if and only if $\boldsymbol{v} \in \overline{\mathrm{NFP}(P_k(o_k) \oplus \boldsymbol{x}_k, P_i(o))}$, and $P_i(o) \oplus \boldsymbol{v}$ does not protrude from the container C if and only if $\boldsymbol{v} \in \overline{\mathrm{NFP}(\overline{C}, P_i(o))}$. Hence, for $\hat{o}$, the set Z of all points $\boldsymbol{v}$ such that $P_i(\hat{o}) \oplus \boldsymbol{v}$ neither overlaps with the other polygon nor protrudes from the container C is given by

$$Z = \bigcap_{P_k \in \mathcal{P} \setminus \{P_i\}} \overline{\mathrm{NFP}(P_k(o_k) \oplus \boldsymbol{x}_k, P_i(\hat{o}))} \cap \overline{\mathrm{NFP}(\overline{C}, P_i(\hat{o}))}.$$

We see that Z is closed and bounded because $\overline{\mathrm{NFP}(P_k(o_k) \oplus \boldsymbol{x}_k, P_i(\hat{o}))}$ ($P_k \in \mathcal{P} \setminus \{P_i\}$) and $\overline{\mathrm{NFP}(\overline{C}, P_i(\hat{o}))}$ are all closed, and $\overline{\mathrm{NFP}(\overline{C}, P_i(\hat{o}))}$ is bounded. Since Z is not empty by the assumption, $\partial Z \subseteq Z$ is not empty either. Z is surrounded by some edges of $\partial\,\mathrm{NFP}(P_k(o_k) \oplus \boldsymbol{x}_k, P_i(\hat{o}))$ and $\partial\,\mathrm{NFP}(\overline{C}, P_i(\hat{o}))$. Therefore, ∂Z includes a point in $\mathcal{V}(o) \cup \mathcal{I}(o)$ and hence FindBestPosition will choose such a point.

For any $(\hat{\boldsymbol{v}}, \hat{o})$ such that $P_i(\hat{o}) \oplus \hat{\boldsymbol{v}}$ neither overlaps with the other polygons nor protrudes from the container, $F(\boldsymbol{x}, \boldsymbol{o})$ is strictly smaller than any other $(\boldsymbol{v}, o)$ such that $P_i(o) \oplus \boldsymbol{v}$ overlaps with another polygon or protrudes from the container, since $F(\boldsymbol{x}, \boldsymbol{o})$ at $(\hat{\boldsymbol{v}}, \hat{o})$ is the sum of the amount of overlap and protrusion of all polygons except P_i. Therefore FindBestPosition will output such a $(\hat{\boldsymbol{v}}, \hat{o})$ if any. □

However, FindBestPosition may not find the globally optimal position if there is no position whose overlap is zero.

2.6.2 Simplifying FindBestPosition

We observed through preliminary experiments that it is time consuming to compute the objective function $F(\boldsymbol{x}, \boldsymbol{o})$ of (2.3) for all points in $\mathcal{V}(o)$ and $\mathcal{I}(o)$ in FindBestPosition. We therefore simplify FindBestPosition by reducing the candidates of $\boldsymbol{v}$ for which we compute $F(\boldsymbol{x}, \boldsymbol{o})$. FindBestPosition2 is the simplified version of FindBestPosition. It checks all possible orientations $o \in O_i$, but it checks them for a prescribed number K of points randomly chosen from $\mathcal{V}(o)$ and for no point in $\mathcal{I}(o)$. The algorithm FindBestPosition2 is formally described in Algorithm 4. FindBestPosition2 does not satisfy Lemma 1.

2.6.3 Swapping Two Polygons

SwapTwoPolygons$(\mathcal{P}, \mathcal{O}, C, \boldsymbol{x}, \boldsymbol{o}, P_i, P_j)$ is an algorithm to swap two polygons P_i and P_j. We designed three ways of swapping, which are described by a parameter $\sigma \in \{\mathtt{C}, \mathtt{I}\} \cup \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. When $\sigma = \mathtt{C}$, we swap the centroids of P_i and P_j (it is only for comparisons in Section 2.7.2). When $\sigma = \mathtt{I}$, we first remove the polygon P_i from the container C, which results in making a hole in the layout. We next place a new

Algorithm 4 : FindBestPosition2($\mathcal{P}, \mathcal{O}, C, \boldsymbol{x}, \boldsymbol{o}, P_i$)

```
F* := +∞;
for each o ∈ O_i do
  Compute NFP(P_k ⊕ x_k, P_i(o)) (P_k ∈ P \ {P_i}) and NFP(C̄, P_i(o));
  Let V be a set of K vertices randomly chosen from V(o);
  for each v ∈ V do
    x' := x;  (x')_i := v;
    o' := o;  (o')_i := o;
    if F(x', o') < F* then
      F* := F(x', o');  v* := v;  o* := o
    end if
  end for
end for;
Return (v*, o*).
```

polygon $P'(o_j) \oplus \boldsymbol{x}_j$, where $P' = P_j$, to prevent P_j from staying at the same place. Then we move P_j to the position computed by FindBestPosition($\mathcal{P}, \mathcal{O}, C, \boldsymbol{x}, \boldsymbol{o}, P_j$) and remove P', where we expect that P_j moves into the hole. Finally, we place the removed polygon P_i by FindBestPosition. When $\sigma \in \mathbb{N}$, we move polygons as the case of $\sigma = \texttt{I}$, but we compute the positions of P_i and P_j by FindBestPosition2($\mathcal{P}, \mathcal{O}, C, \boldsymbol{x}, \boldsymbol{o}, P_j$), where we set $K = \sigma$. The algorithm SwapTwoPolygons is formally described in Algorithm 5, where $(\mathcal{P})_i$ denotes the ith element of $\mathcal{P}$ and $(\mathcal{O})_i$ denotes the ith element of $\mathcal{O}$. We compare the three ways of swapping by computational experiments in Section 2.7.2.

2.6.4 Initial Solution of ILSQN

We generate an initial feasible layout of ILSQN using FindBestPosition or FindBestPosition2. We assume that the initial length L of the container $C(W, L)$ is large enough to place all polygons in $\mathcal{P}$ without overlap in C. We prepare a sequence of polygons in $\mathcal{P}$ and place polygons one by one in the order of the sequence, where the position of each polygon P_i is decided by FindBestPosition or FindBestPosition2. We control the sequence by using a parameter $S_{\text{init}} \in \{\texttt{sort}, \texttt{random}\}$: the sequence of the descending order of area if $S_{\text{init}} = \texttt{sort}$ and a random sequence if $S_{\text{init}} = \texttt{random}$. We use FindBestPosition if $\sigma \in \{\texttt{C}, \texttt{I}\}$ and FindBestPosition2 if $\sigma \in \mathbb{N}$, where we give σ to FindBestPosition2 as the parameter K. If there is more than one position with no overlap, we choose the bottom-left position (i.e., the position with the minimum x_{i1}, breaking ties with the minimum x_{i2}, where $\boldsymbol{x}_i = (x_{i1}, x_{i2})$ is the translation vector of polygon P_i). We compute the length L of the container by (2.2) after placing all polygons in $\mathcal{P}$.

There are variations in choosing a sequence of polygons such as the descending order of

Algorithm 5 : SwapTwoPolygons($\mathcal{P}, \mathcal{O}, C, \boldsymbol{x}, \boldsymbol{o}, P_i, P_j$)

/* For $\sigma = \mathtt{C}$ */
for each $(o'_i, o'_j) \in O_i \times O_j$ **do**
  Swap $P_i(o'_i)$ and $P_j(o'_j)$ at their centroids;
  Let $(\boldsymbol{x}', \boldsymbol{o}')$ be the resulting layout
end for;
Let $(\boldsymbol{x}, \boldsymbol{o})$ be the $(\boldsymbol{x}', \boldsymbol{o}')$ with the minimum $F(\boldsymbol{x}', \boldsymbol{o}')$ among those generated in the above loop;
Return $(\boldsymbol{x}, \boldsymbol{o})$.

/* For $\sigma = \mathtt{I}$ */
$\mathcal{P}' := \mathcal{P}$; $\mathcal{O}' := \mathcal{O}$; $\boldsymbol{x}' := \boldsymbol{x}$; $\boldsymbol{o}' := \boldsymbol{o}$;
$(\mathcal{P}')_i := (\mathcal{P})_j$; $(\mathcal{O}')_i := (\boldsymbol{o})_j$; $(\boldsymbol{x}')_i := (\boldsymbol{x})_j$; $(\boldsymbol{o}')_i := (\boldsymbol{o})_j$;
$((\boldsymbol{x})_j, (\boldsymbol{o})_j) :=$ FindBestPosition$(\mathcal{P}', \mathcal{O}', C, \boldsymbol{x}', \boldsymbol{o}', (\mathcal{P}')_j)$;
$((\boldsymbol{x})_i, (\boldsymbol{o})_i) :=$ FindBestPosition$(\mathcal{P}, \mathcal{O}, C, \boldsymbol{x}, \boldsymbol{o}, (\mathcal{P})_i)$;
Return $(\boldsymbol{x}, \boldsymbol{o})$.

/* For $\sigma \in \mathbb{N}$ */
$\mathcal{P}' := \mathcal{P}$; $\mathcal{O}' := \mathcal{O}$; $\boldsymbol{x}' := \boldsymbol{x}$; $\boldsymbol{o}' := \boldsymbol{o}$;
$(\mathcal{P}')_i := (\mathcal{P})_j$; $(\mathcal{O}')_i := (\boldsymbol{o})_j$; $(\boldsymbol{x}')_i := (\boldsymbol{x})_j$; $(\boldsymbol{o}')_i := (\boldsymbol{o})_j$;
$((\boldsymbol{x})_j, (\boldsymbol{o})_j) :=$ FindBestPosition2$(\mathcal{P}', \mathcal{O}', C, \boldsymbol{x}', \boldsymbol{o}', (\mathcal{P}')_j)$;
$((\boldsymbol{x})_i, (\boldsymbol{o})_i) :=$ FindBestPosition2$(\mathcal{P}, \mathcal{O}, C, \boldsymbol{x}, \boldsymbol{o}, (\mathcal{P})_i)$;
Return $(\boldsymbol{x}, \boldsymbol{o})$.

Table 2.1: The information of the benchmark instances for the two-dimensional irregular strip packing problem (cited from [45])

Instance	NDP	TNP	ANV	Orientations (°)	Width
ALBANO	8	24	7.25	0, 180	4900
DAGLI	10	30	6.30	0, 180	60
DIGHE1	16	16	3.87	0	100
DIGHE2	10	10	4.70	0	100
FU	12	12	3.58	0, 90, 180, 270	38
JAKOBS1	25	25	5.60	0, 90, 180, 270	40
JAKOBS2	25	25	5.36	0, 90, 180, 270	70
MAO	9	20	9.22	0, 90, 180, 270	2550
MARQUES	8	24	7.37	0, 90, 180, 270	104
SHAPES0	4	43	8.75	0	40
SHAPES1	4	43	8.75	0, 180	40
SHAPES2	7	28	6.29	0, 180	15
SHIRTS	8	99	6.63	0, 180	40
SWIM	10	48	21.90	0, 180	5752
TROUSERS	17	64	5.06	0, 180	79

NDP: The number of different polygons
TNP: The total number of polygons
ANV: The average number of vertices of different polygons

area and a random order. Gomes and Oliveira [45] also generated initial solutions in a similar way using a different sequence of polygons.

2.7 Experimental Results

This section reports the results on computational experiments of our algorithm ILSQN and other algorithms.

2.7.1 Environment

Benchmark instances for the irregular strip packing problem are available online at EURO Special Interest Group on Cutting and Packing (ESICUP) website.[1] Table 2.1 shows the information of the instances. The column NDP denotes the number of different polygons, the column TNP shows the total number of polygons, the column ANV shows the average number of the vertices of different polygons, the column Orientations represents the permitted orientations, and the column Width shows the width W of the container.

[1]ESICUP: `http://www.fe.up.pt/esicup/`

We implemented our algorithm ILSQN in C++, compiled it by GCC 4.0.2 and conducted computational experiments on a PC with an Intel Xeon 2.8GHz processor and 1GB memory. We adopted a quasi-Newton method package L-BFGS [72] for algorithm SEPARATE. L-BFGS has a parameter m_{BFGS} that is the number of BFGS corrections in L-BFGS. We set $m_{\mathrm{BFGS}} = 6$ because $3 \leq m_{\mathrm{BFGS}} \leq 7$ is recommended in the L-BFGS package.

A layout is judged to be feasible when the objective function $F(\boldsymbol{x}, \boldsymbol{o})$ of (2.3) is less than $\epsilon = 10^{-10}W^2$ due to limited precision. Thus, our algorithm may generate layouts that have a slight overlap.

2.7.2 Parameters

ILSQN has the following parameters:

- $r_{\mathrm{dec}} \in (0, 1)$: the ratio by which ILSQN shortens the container.
- $r_{\mathrm{inc}} \in (0, 1)$: the ratio by which ILSQN extends the container.
- $\pi_{\mathrm{side}} \in \{\texttt{left}, \texttt{right}, \texttt{both}\}$: the side of the container ILSQN shortens or extends.
- $N_{\mathrm{mo}} > 0$: the termination criterion of MINIMIZEOVERLAP.
- $S_{\mathrm{init}} \in \{\texttt{sort}, \texttt{random}\}$: the sequence from which ILSQN generates an initial solution. "$S_{\mathrm{init}} = \texttt{sort}$" means the descending order of area and "$S_{\mathrm{init}} = \texttt{random}$" means a random sequence.
- $\sigma \in \{\texttt{C}, \texttt{I}\} \cup \mathbb{N}$: the ways of generating an initial solution in ILSQN and swapping two polygons in SWAPTWOPOLYGONS.

  In order to generate an initial solution, ILSQN uses FINDBESTPOSITION if $\sigma = \texttt{I}$ or $\texttt{C}$, and uses FINDBESTPOSITION2 if $\sigma \in \mathbb{N}$, where $\sigma \in \mathbb{N}$ is the parameter K of FINDBESTPOSITION2.

  SWAPTWOPOLYGONS swaps the centroids of two given polygon if $\sigma = \texttt{C}$, swaps them by FINDBESTPOSITION if $\sigma = \texttt{I}$, and swaps them by FINDBESTPOSITION2 if $\sigma \in \mathbb{N}$, where $\sigma \in \mathbb{N}$ is the parameter K of FINDBESTPOSITION2.

We measure the *efficiency* of a solution by the ratio

$$\frac{\text{the total area of the polygons}}{\text{the area of the container}}.$$

We investigated the effects of the parameters by the following four computational experiments. We chose six benchmark instances (SHAPES0, SHAPES1, SHAPES2, SHIRTS, SWIM, TROUSERS), and conducted 5 runs with the time limit of each run being 10 minutes.

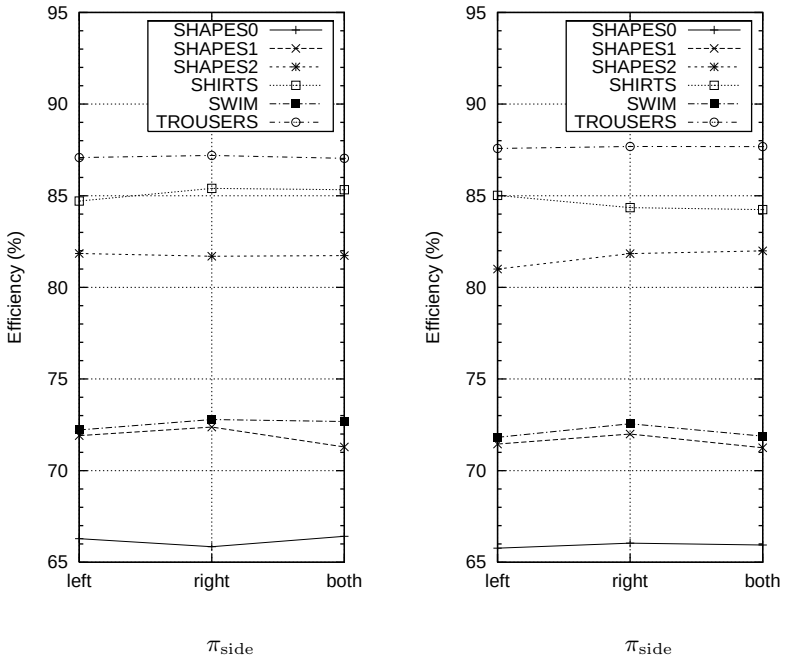


Figure 2.12: The average efficiencies against π_{side} (left: $S_{\mathrm{init}} = \texttt{sort}$, right: $S_{\mathrm{init}} = \texttt{random}$)

We first examined the effect of π_{side}. Figure 2.12 shows the average efficiency of five runs; the graph on the left shows the result of $S_{\mathrm{init}} = \texttt{sort}$, and the one on the right shows the result of $S_{\mathrm{init}} = \texttt{random}$, where we set $r_{\mathrm{dec}} = 0.04, r_{\mathrm{inc}} = 0.01, N_{\mathrm{mo}} = 200$, and $\sigma = \texttt{I}$. These results indicate that π_{side} does not have much effect on the efficiency.

We second examined the effect of r_{dec} and r_{inc}. Figure 2.13 shows the average efficiency of five runs; the graph on the left shows the result of $S_{\mathrm{init}} = \texttt{sort}$, and the one on the right shows the result of $S_{\mathrm{init}} = \texttt{random}$, where we set $N_{\mathrm{mo}} = 200$, $\pi_{\mathrm{side}} = \texttt{right}$, $\sigma = \texttt{I}$. From these results, we observe that the efficiency is almost same for $r_{\mathrm{dec}} \leq 0.06$.

Next, we investigated the effect of N_{mo}. Figure 2.14 shows the average efficiency of five runs; the graph on the left shows the result of $S_{\mathrm{init}} = \texttt{sort}$, and the one on the right shows the result of $S_{\mathrm{init}} = \texttt{random}$, where we set $r_{\mathrm{dec}} = 0.04, r_{\mathrm{inc}} = 0.01$, $\pi_{\mathrm{side}} = \texttt{right}$, and $\sigma = \texttt{I}$. These results indicate that the efficiency is slightly better for $50 \leq N_{\mathrm{mo}} \leq 300$.

Finally, we checked the effect of σ. Figure 2.15 shows the average efficiency of five runs; the graph on the left shows the result of $S_{\mathrm{init}} = \texttt{sort}$, and the one on the right shows the result of $S_{\mathrm{init}} = \texttt{random}$, where we set $r_{\mathrm{dec}} = 0.04, r_{\mathrm{inc}} = 0.01$, $\pi_{\mathrm{side}} = \texttt{right}$, and $N_{\mathrm{mo}} = 200$. These results indicate that the efficiency is clearly worse for $\sigma = \texttt{C}$, the efficiency is better for $100 \leq \sigma \leq 800$, and the best result is obtained when $S_{\mathrm{init}} = \texttt{sort}$ and $\sigma = 800$. We can also observe from Figures 2.12–2.15 that S_{init} does not have much effect on the efficiency.

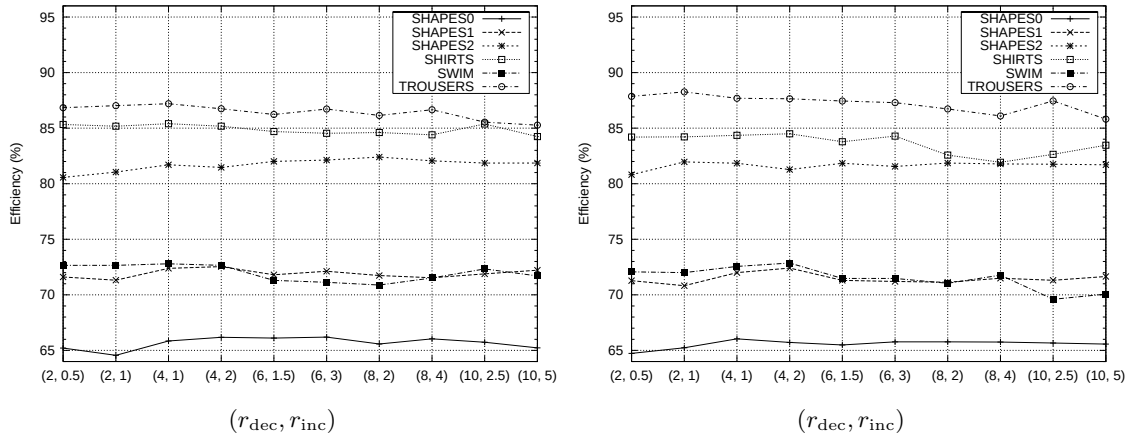


Figure 2.13: The average efficiencies against $(r_{\mathrm{dec}}, r_{\mathrm{inc}})$ in % (left: $S_{\mathrm{init}} =$ `sort`, right: $S_{\mathrm{init}} =$ `random`)

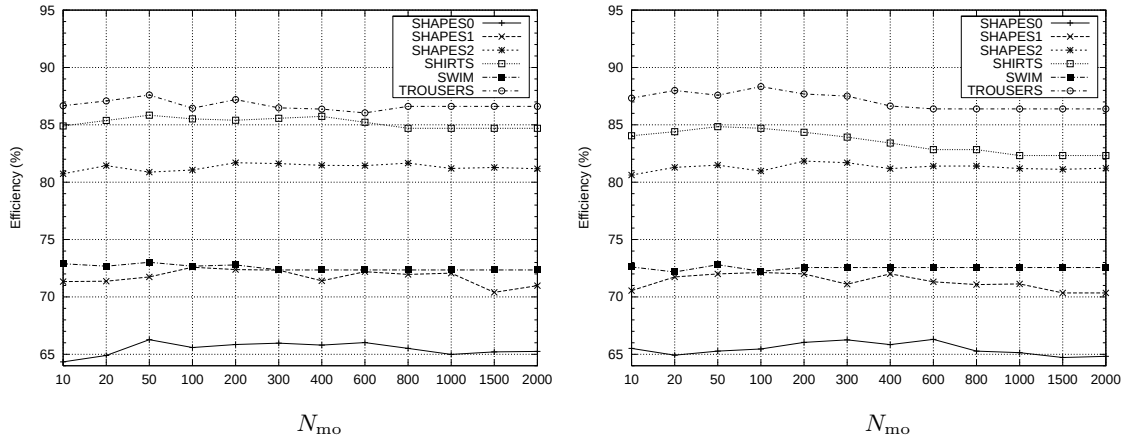


Figure 2.14: The average efficiencies against N_{mo} (left: $S_{\mathrm{init}} =$ `sort`, right: $S_{\mathrm{init}} =$ `random`)

Based on these observations, we set $S_{\mathrm{init}} =$ `sort`, $\pi_{\mathrm{side}} =$ `right`, $r_{\mathrm{dec}} = 0.04$, $r_{\mathrm{inc}} = 0.01$, $N_{\mathrm{mo}} = 200$, and $\sigma = 800$ for the computational experiments of all benchmark instances in Section 2.7.3.

2.7.3 Results

In this subsection, we show the computational results of our algorithm ILSQN comparing it with other existing algorithms. We ran algorithm ILSQN ten times for each instances listed in Table 2.1 and compared our results with those reported by Gomes and Oliveira [45] (denoted as "SAHA"), Burke et al. [23] (denoted as "BLF") and Egeblad et al. [32] (denoted as "2DNest"). Table 2.2 shows the best and average length and efficiency in % of ILSQN and the best efficiency in % of the other algorithms. The column EF shows the efficiency in %.

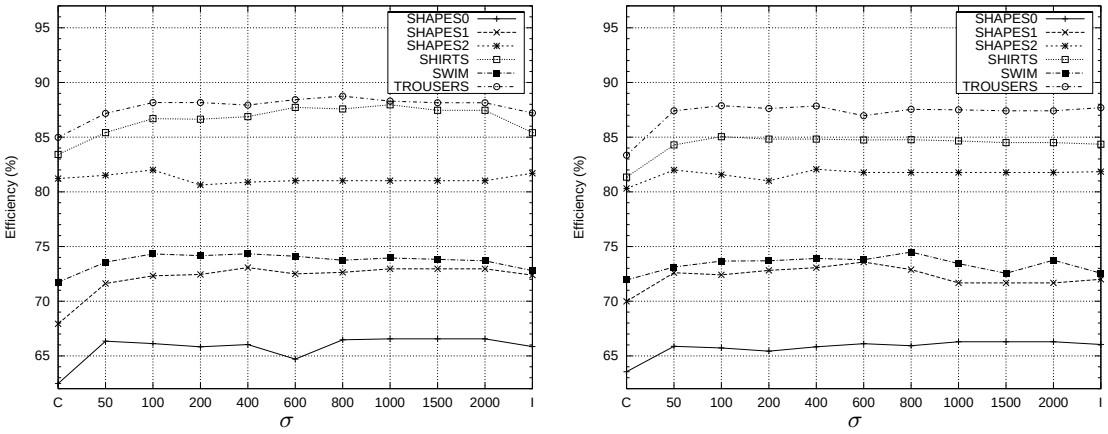


Figure 2.15: The average efficiencies against σ (left: $S_{\text{init}} = \texttt{sort}$, right: $S_{\text{init}} = \texttt{random}$)

The best results among these algorithms are written in bold typeface. Table 2.3 shows the computation time (in seconds) of the algorithms.

Gomes and Oliveira [45] did not use time limit but stopped their algorithm by other criteria. They conducted 20 runs for each instance and the best results of the 20 runs are shown in Table 2.2, while their computation times in Table 2.3 are the average computation time of the 20 runs.

Burke et al. [23] tested four variations of their algorithm, and conducted 10 runs for each variation. Their results in Table 2.2 are the best results of the 40 runs, which are taken from Table 5 in [23]. They limited the number of iterations for each run, and their computation time in Table 2.3 is the time spent to find the best solution reported in Table 2.2 in the run that found it (i.e., the time for only one run is reported). Since they conducted experiments for instances ALBANO, DIGHE1 and DIGHE2 with different orientations from the others, we do not include the results.

Egeblad et al. [32] and we conducted experiments using the time limits for each run shown in Table 2.3.

Although our total computation time of all runs for each instance is not so long compared with SAHA [45] and 2DNest [32], ILSQN obtained the best results for 8 instances out of the 15 instances in efficiency of the resulting layouts and also obtained the results with almost equivalent efficiency to the best results for some instances. It achieved the same efficiency as 2DNest [32] for instance SHAPES1, but the layouts are different. The computation time of BLF [23] is much shorter than that of ILSQN, and ILSQN obtained better results in efficiency than those BLF [23] obtained for all instances. Figures 2.16 and 2.17 show the best layouts obtained by ILSQN for all instances.

Table 2.2: The length of ILSQN and the efficiency in % of the four algorithms

Instance	ILSQN				SAHA [45]	BLF [23]	2DNest [32]
	Average		Best		Best	Best	Best
	Length	EF (%)	Length	EF (%)	EF (%)	EF (%)	EF (%)
ALBANO	9990.23	87.14	9874.48	**88.16**	87.43	–	87.44
DAGLI	59.11	85.80	58.02	**87.40**	87.15	83.7	85.98
DIGHE1	110.69	90.49	100.11	99.89	**100.00**	–	99.86
DIGHE2	120.43	84.21	100.01	99.99	**100.00**	–	99.95
FU	32.56	87.57	31.43	90.67	90.96	86.9	**91.84**
JAKOBS1	11.56	84.78	11.28	86.89	†*78.89	82.6	**89.07**
JAKOBS2	23.98	80.50	23.39	**82.51**	77.28	74.8	80.41
MAO	1813.38	81.31	1766.43	83.44	82.54	79.5	**85.15**
MARQUES	79.72	86.81	77.70	89.03	88.14	86.5	**89.17**
SHAPES0	60.02	66.49	58.30	**68.44**	66.50	60.5	67.09
SHAPES1	54.79	72.83	54.04	**73.84**	71.25	66.5	**73.84**
SHAPES2	26.44	81.72	25.64	**84.25**	83.60	77.7	81.21
SHIRTS	61.28	88.12	60.83	**88.78**	†86.79	84.6	86.33
SWIM	5928.23	74.62	5875.17	**75.29**	74.37	68.4	71.53
TROUSERS	245.58	88.69	242.56	89.79	**89.96**	88.5	89.84

* The value has been corrected from the one reported in [45] according to the information sent from the authors of [45].

† Better results were obtained by a simpler greedy approach (GLSHA)[45]: 81.67% for JAKOBS1 and 86.80% for SHIRTS.

Table 2.3: The computation time in seconds of the four algorithms

Instance	*ILSQN Xeon 2.8 GHz 10 runs	†SAHA [45] Pentium4 2.4 GHz 20 runs	‡BLF [23] Pentium4 2.0 GHz 4×10 runs	*2DNest [32] Pentium4 3.0 GHz 20 runs
ALBANO	1200	2257	–	600
DAGLI	1200	5110	188.80	600
DIGHE1	600	83	–	600
DIGHE2	600	22	–	600
FU	600	296	20.78	600
JAKOBS1	600	332	43.49	600
JAKOBS2	600	454	81.41	600
MAO	1200	8245	29.74	600
MARQUES	1200	7507	4.87	600
SHAPES0	1200	3914	21.33	600
SHAPES1	1200	10314	2.19	600
SHAPES2	1200	2136	21.00	600
SHIRTS	1200	10391	58.36	600
SWIM	1200	6937	607.37	600
TROUSERS	1200	8588	756.15	600

* Computation time is the time limit for each run.

† Computation time is the average computation time.

‡ Computation time is the time spent to find the best solution in the run that found it.

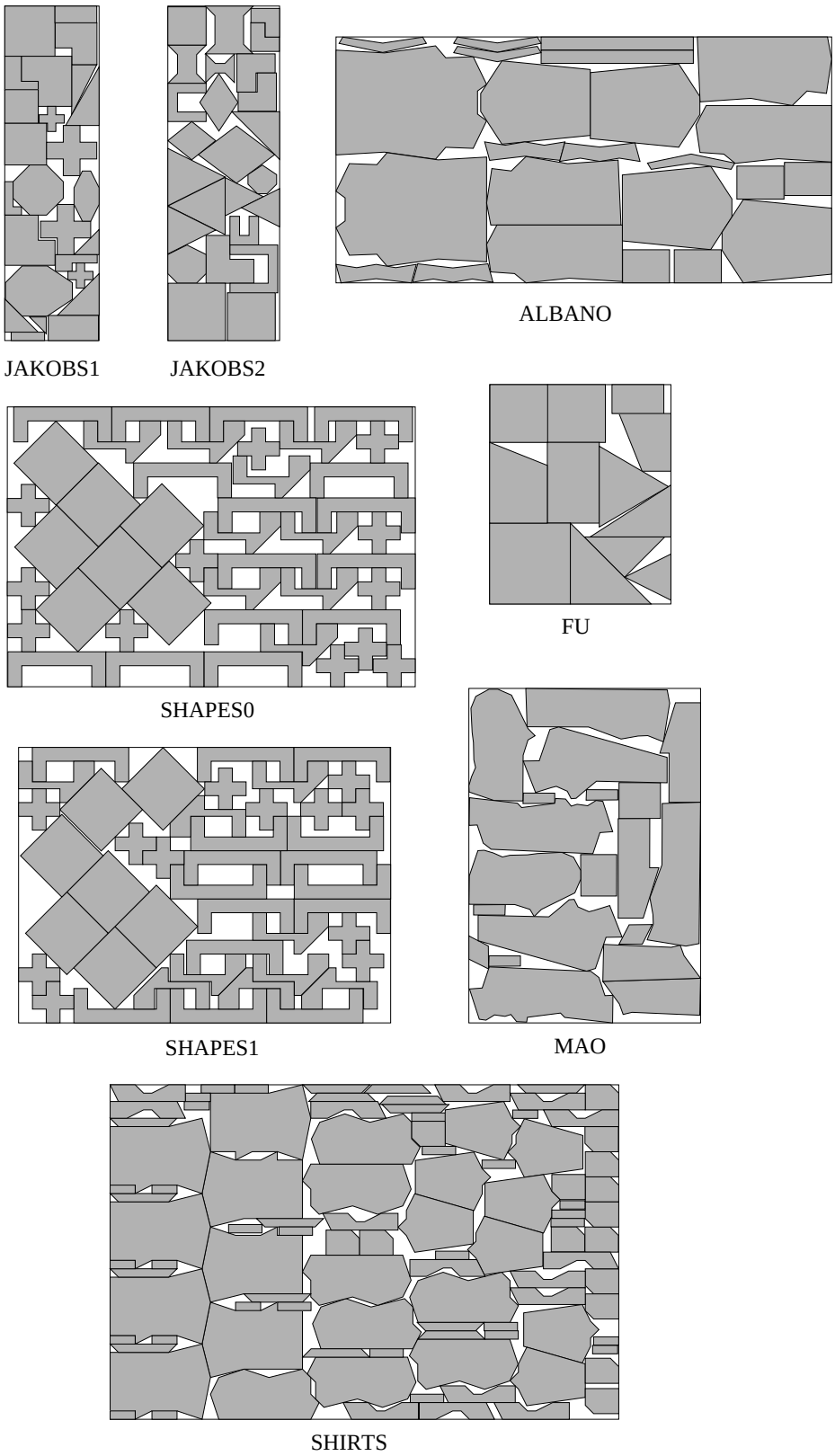
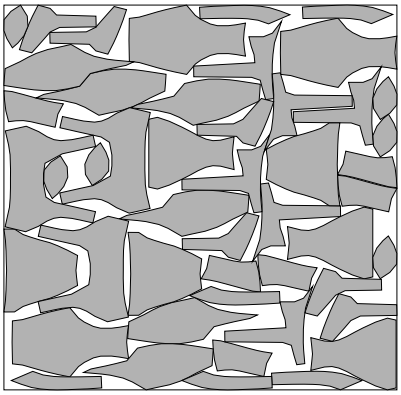
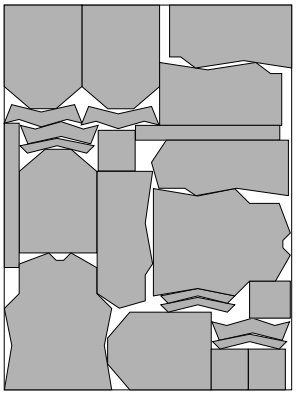


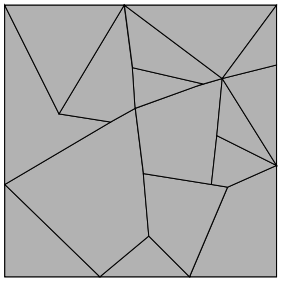
Figure 2.16: The best solutions for JAKOBS1, JAKOBS2, ALBANO, FU, SHAPES0, SHAPES1, MAO and SHIRTS obtained by ILSQN



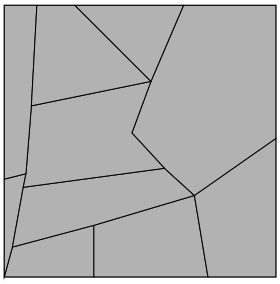
SWIM



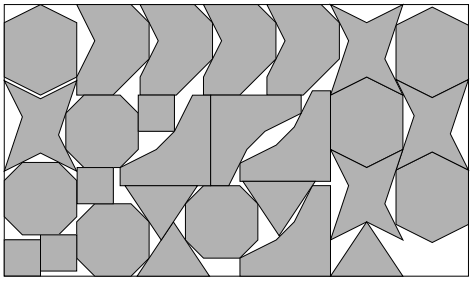
MARQUES



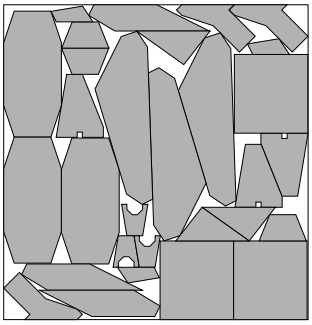
DIGHE1



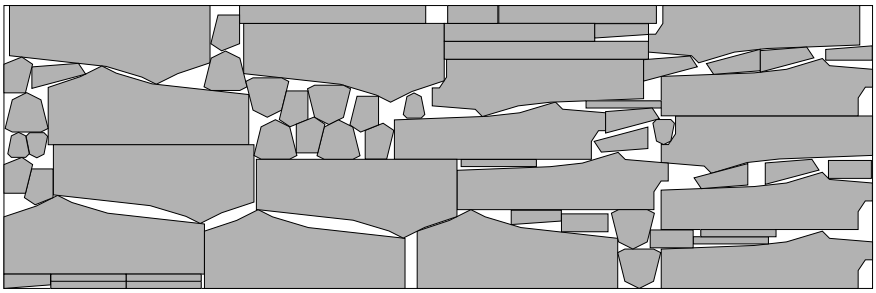
DIGHE2



SHAPES2



DAGLI



TROUSERS

Figure 2.17: The best solutions for SWIM, MARQUES, DIGHE1, DIGHE2, SHAPES2, DAGLI and TROUSERS obtained by ILSQN

2.8 Summary

We proposed an iterated local search algorithm for the overlap minimization problem consolidating a separation algorithm based on nonlinear programming and a swapping operation of two polygons, and incorporated it into our algorithm ILSQN for the irregular strip packing problem. We showed through computational experiments that ILSQN is competitive with existing algorithms, updating the best known solutions for several benchmark instances.

It is left as future work to combine our algorithm and the multi-stage approach [45] to obtain better solutions quickly. For this approach, we need to develop an algorithm to automatically cluster polygons by some criteria. Computation of a good lower bound of the irregular strip packing problem is also important to evaluate heuristic algorithms or to develop exact algorithms.

Chapter 3

Multi-sphere Scheme

3.1 Introduction

A packing problem requires a given set of objects such as non-convex polytopes or objects with curved surfaces to be placed compactly in a bounded space in $\mathbb{R}^2$ and $\mathbb{R}^3$, where we treat translation, rotation and deformation as possible motions of each object. As we explained in Section 2.3.1, the no-fit polygon is a useful tool to check whether two non-convex polygons overlap or not and to compute their penetration depth quickly. However, in general the robust implementation of computation of no-fit polygons is not an easy task especially in $\mathbb{R}^3$, and moreover if free rotation is allowed in motions of polygons, then no-fit polygons cannot be used directly.

To overcome this difficulty, we treat a given object as a set of spheres which gives approximately the same boundary of the original object. The reason why we represent a given object with a set of spheres is that geometric computations such as collision detection and penetration depth become significantly simpler if all objects are restricted to be spheres. Based on this fact, in this chapter, we propose a general approach, called *multi-sphere scheme* to provide a basis for designing efficient algorithms for various types of packing problems.

Structure of the Multi-sphere Scheme. The scheme consists of three procedures: *approximation*, *local search* and *global search* as its building blocks. Figure 3.1 illustrates the relationships between the procedures, where boxes with solid lines denote these operations.

- Approximation: Approximation generates a set of spheres that represents each object of a given instance. For an object O in 2D (resp., 3D), a set $\mathcal{S}(O)$ of circles (resp., spheres) is computed so that all the circles (resp., spheres) are contained in the interior of the original instance, the area covered by the circles (resp., spheres) is maximized, and the boundary of the union of the circles (resp., spheres) approximates that of

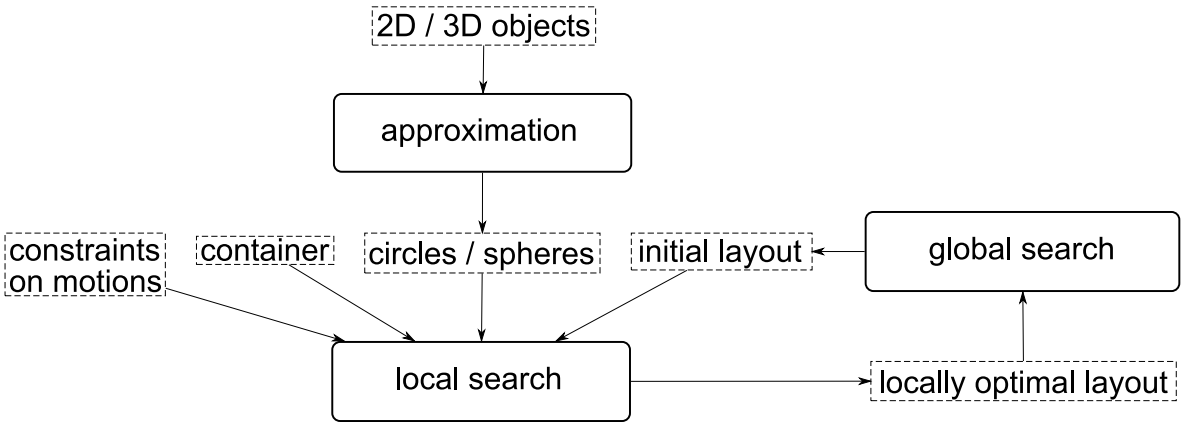


Figure 3.1: The structure of the multi-sphere scheme

the original object as precise as possible. If we introduce a sufficiently many number of circles/spheres, then the above approximation is not difficult to achieve. However, as we examine the computational burden due to many circles/spheres in Sections 3.3 and 3.4. it is an important issue to determine an appropriate number of circles/spheres to be used in approximation. A container can also be represented by a set of spheres.

An object is called *rigid* if it is not allowed to be transformed, and is called *deformable* otherwise. If a rigid object O in 3D is be represented by a set $\mathcal{S}(O)$ of spheres, then we need to store the relative positions among the spheres in $\mathcal{S}(O)$ and let the spheres in $\mathcal{S}(O)$ meet the positions in a final packing layout. Note that, for some applications such as protein packing, a given object is a union of spheres, and in such a case, we only need to maintain the relative positions among the spheres.

- Local Search: The input for local search in the multi-sphere scheme consists of objects to be placed in 2D or 3D, a container, constraints on motions of objects, and an initial layout of the objects, where each object O is given as a set $\mathcal{S}(O)$ of spheres. No container may be given in some application. In this case, we introduce a container with adequate size and shape. Local search is a procedure that improves the layout by a continuous sequence of movements of the objects. For this, we impose a penalty on two intersecting objects (which are not allowed to intersect each other) and on an object that protrudes the container. No penalty is imposed on two intersecting spheres in the same sphere set $\mathcal{S}(O)$ unless a special problem setting is considered. The penalty is represented by a penalty function which is defined based on the penetration depth of spheres. Local search is designed as an algorithm for solving an unconstrained nonlinear programming problem that minimizes the penalty function, where the given layout corresponds to an

initial solution to the nonlinear programming problem while a local optimal solution to the nonlinear programming problem corresponds to a final layout of the objects. Specified constrains on motions of objects (if any) are formulated part of the nonlinear programming problem. Since we exploit an iterative algorithm such as quasi-Newton method to solve the the nonlinear programming problem, the iteration by the algorithm gives a sequence of movements of the objects in the corresponding layout, which usually can be regarded as a continuous sequence of movements of objects that reduce the overlap gradually. Note that a local optimal solution to the nonlinear programming problem may give a final layout in which some object still intersects with other objects or the container. In such a case, the given instance has no overlap-free layout or the initial layout needs to be changed to find an overlap-free layout.

- Global Search: To seek for a layout of given objects with no penetration and no protrusions, global search moves objects globally in the following sense. Global search is a procedure that constructs an initial layout of a given layout and modifies final layouts by the local search into a new initial layout for the local search. To achieve this, we utilize local search and other operations of objects such as swapping two objects to obtain better initial layouts.

The multi-sphere scheme is so flexible that it can deal with various types of packing problems. For example, we show two representative cases.

- Suppose that we want to find a layout with no penetrations and no protrusions of an instance of a problem, which includes rigid objects, a container and constraints on motions (no initial layout), we first apply the approximation to each object and obtain a set of spheres. Then we apply the global search to the sets of spheres, the container and the constraints on motions. (the local search may be invoked in the global search). See Section 3.2.3 for an example of the global search. This is for packing 2D and 3D objects with translation and rotation. See Section 3.4.6 for the results.

- If we want to improve a given layout of rigid objects, where a container is given and constraints on motions of the objects are predefined, we first apply the approximation to the objects and then apply the local search to obtain a locally optimal layout. See Section 3.5. We apply the scheme to problems to remove overlaps of labels, which arise from Graph Drawing.

Throughout the chapter, we discuss local search algorithms for the problem of modifying a given layout of objects to reduce the amount of overlap in the layout, where each object is

rigid and any two intersecting objects are penalized. Designing algorithms for other variations of problem settings such as deformable objects will be discussed as future work in Chapter 4.

Related Work. Modelling with spheres plays an important role in a numerous fields such as computer graphics [83], physical simulations [3, 40, 92], bioinformatics [28, 46], and packing problems [19, 93] and collision detection of spheres is critical in the efficiency of algorithms based on modelling with spheres.

Modeling an object by a sphere set is already used in several applications. Ferrez [35] proposed a framework of physical simulation of a set of spheres in $\mathbb{R}^3$ and a fast algorithm to detect the collisions of spheres. Hubbard [55] proposed a method that approximates an object by a hierarchy of spheres to accelerate the collision detections of the objects. Agarwal et al. [3] proposed the *deformable necklace* model that is a flexible chain of spheres in which only adjacent spheres may intersect. They studied bounding volume hierarchy based on spheres for collision and self-collision detection.

Organization. This chapter is organized as follows. In Section 3.2, we propose a local search algorithm and an iterated local search algorithm. In Section 3.3, we propose a collision detection algorithm of spheres and analyze the performance theoretically. In Section 3.4, we show how to implement the collision detection algorithm into the multi-sphere scheme and the computation results. Finally, in Section 3.5 we apply the local search algorithm of the multi-sphere scheme to two different kinds of label overlap removal and show the computation results.

3.2 Algorithms for Multi-sphere Scheme

In this section, we focus on the *penalized rigid sphere set packing problem* that asks to pack a family of sphere sets in a container. The container is a rectangle for $\mathbb{R}^2$ and a cuboid for $\mathbb{R}^3$. Our approach allows any object to intersect with other object or protrude from the container during a modification of the layout of objects. For this, we penalize the amount of penetration and protrusion. The amount of penetration and protrusion is defined in such a way that a solution with penalty zero gives a layout with no intersection, and the sum of all penalties is represented by a differentiable function. We formulate the problem as a nonlinear program, and propose a local search algorithm RigidQN that finds a local optimal layout from an initial layout by applying quasi-Newton method to the penalized rigid sphere set packing problem.

In this section, we consider three variations of motions of objects: "translation," "trans-

lation with a fixed direction," and "translation and rotation by an arbitrary angle." Our exposition mainly focuses on the case for $\mathbb{R}^3$, from which the argument for $\mathbb{R}^2$ can be easily derived.

Especially for the case where objects are allowed to translate and rotate, we incorporate RigidQN and a swapping perturbation of two objects to obtain our iterated local search algorithm ILS_Rigid. We implement ILS_Rigid and conduct computational experiments on some instances of two different problems. Computational experiments reveal our algorithm can find a layout with nearly no penalty quickly.

3.2.1 Penalized Rigid Sphere Set Packing Problem

In this section, we formulate the penalized rigid sphere set packing problem for $\mathbb{R}^d$, which asks to move a collection $\mathcal{O} = \{O_1, \ldots, O_m\}$ of m objects so that no two objects overlap each other. Each object O_i consists of n_i spheres $\{S_{i1}, \ldots, S_{in_i}\}$. Let $\boldsymbol{c}_{ij}$ be the vector that represents the center of spheres S_{ij}, r_{ij} be the radius of S_{ij} and $N = \sum_{i=1}^{m} n_i$ be the total number of spheres. We let $\boldsymbol{r}_i = \sum_{j=1}^{n_i} \boldsymbol{c}_{ij}/n_i$, which represents the center of O_i. After translating object O by a translation vector $\boldsymbol{v} \in \mathbb{R}^d$, the resulting object is described as $O \oplus \boldsymbol{v} = \{\boldsymbol{x} + \boldsymbol{v} \mid \boldsymbol{x} \in O\}$.

Let $\Lambda_i(\boldsymbol{x}, \boldsymbol{v}) : \mathbb{R}^{d \times \lambda_i} \to \mathbb{R}^d$ $(i = 1, \ldots, m)$ be a motion function that moves a point $\boldsymbol{x} \in \mathbb{R}^d$ by λ_i variables $\boldsymbol{v} \in \mathbb{R}^{\lambda_i}$. For a set of points $S \subseteq \mathbb{R}^d$, $\Lambda_i(S, \boldsymbol{v}) = \{\Lambda_i(\boldsymbol{x}, \boldsymbol{v}) \mid \boldsymbol{x} \in S\}$. For simplicity, we let $\boldsymbol{c}_{ij}(\boldsymbol{v}) = \Lambda_i(\boldsymbol{c}_{ij}, \boldsymbol{v})$ and $S_{ij}(\boldsymbol{v}) = \Lambda_i(S_{ij}, \boldsymbol{v})$.

For spheres S_{ij} and S_{kl}, the penetration depth of them is given by

$$\delta(S_{ij}, S_{kl}) = \max\{r_{ij} + r_{kl} - \|\boldsymbol{c}_{ij} - \boldsymbol{c}_{kl}\|, 0\}.$$

We define penalties of the penetration and protrusion by using the penetration depth. Let

$$f_{ijkl}^{\mathrm{pen}}(\boldsymbol{v}) = [\delta(S_{ij}(\boldsymbol{v}_i), S_{kl}(\boldsymbol{v}_k))]^2$$

be the penetration penalty of two spheres S_{ij} and S_{kl},

$$f_{ij}^{\mathrm{pro}}(\boldsymbol{v}) = [\delta(S_{ij}(\boldsymbol{v}_i), \mathrm{cl}(\overline{C}))]^2$$

be the protrusion penalty of sphere S_{ij} from the container C. Let

$$F_{\mathrm{pen}}(\boldsymbol{v}) = \sum_{1 \le i < k \le m} \sum_{j=1}^{n_i} \sum_{l=1}^{n_k} f_{ijkl}^{\mathrm{pen}}(\boldsymbol{v})$$

be the total penetration penalty, and

$$F_{\mathrm{pro}}(\boldsymbol{v}) = \sum_{i=1}^{m} \sum_{j=1}^{n_i} f_{ij}^{\mathrm{pro}}(\boldsymbol{v})$$

be the total protrusion penalty. Then, the penalized rigid sphere set packing problem for $\mathbb{R}^3$ is defined by

$$\begin{aligned} \text{minimize} \quad & F_{\text{rigid}}(\boldsymbol{v}) = w_{\text{pen}} F_{\text{pen}}(\boldsymbol{v}) + w_{\text{pro}} F_{\text{pro}}(\boldsymbol{v}), \\ \text{subject to} \quad & \boldsymbol{v} = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m) \in \mathbb{R}^{\sum_{i=1}^m \lambda_i}, \\ & \boldsymbol{v}_i \in \mathbb{R}^{\lambda_i}, \quad i = 1, \ldots, m, \end{aligned} \tag{3.1}$$

where w_{pen} and w_{pro} are positive parameters. The penalized rigid sphere set packing problem for $\mathbb{R}^2$ can be formulated analogously.

Computation of the Gradient of the Objective Function

We compute $\nabla F_{\text{rigid}}(\boldsymbol{v})$ by computing all $\nabla f_{ijkl}^{\text{pen}}$ and $\nabla f_{ij}^{\text{pro}}$ and summing them up. We explain how to compute $\nabla f_{ijkl}^{\text{pen}}$ and $\nabla f_{ij}^{\text{pro}}$ in the following subsections.

Intersection of Two Objects. If a pair of spheres has no intersection, the pair does not contribute F_{pen}. Thus, $\partial f_{ijkl}^{\text{pen}}(\boldsymbol{v})/\partial \boldsymbol{v}_a = \boldsymbol{0}$ $(a \in \{1, \ldots, m\} \setminus \{i, k\})$. Assume that a pair of spheres $S_{ij}(\boldsymbol{v}_i)$ and $S_{kl}(\boldsymbol{v}_k)$ intersect each other. Let $(x_{ij}, y_{ij}, z_{ij})^{\mathsf{T}} = \boldsymbol{c}_{ij}(\boldsymbol{v}_i)$. If $\boldsymbol{c}_{ij}(\boldsymbol{v}_i) \neq \boldsymbol{c}_{kl}(\boldsymbol{v}_k)$, then it holds

$$\frac{\partial f_{ijkl}^{\text{pen}}(\boldsymbol{v})}{\partial \boldsymbol{v}_i} = -2\delta(S_{ij}(\boldsymbol{v}_i), S_{kl}(\boldsymbol{v}_k)) \cdot \frac{\partial \boldsymbol{c}_{ij}(\boldsymbol{v}_i)^{\mathsf{T}}}{\partial \boldsymbol{v}_i} \cdot \frac{\boldsymbol{c}_{ij}(\boldsymbol{v}_i) - \boldsymbol{c}_{kl}(\boldsymbol{v}_k)}{\|\boldsymbol{c}_{ij}(\boldsymbol{v}_i) - \boldsymbol{c}_{kl}(\boldsymbol{v}_k)\|},$$

Because $f_{ijkl}^{\text{pen}}(\boldsymbol{v})$ is nondifferentiable in the case of $\boldsymbol{c}_{ij}(\boldsymbol{v}_i) = \boldsymbol{c}_{kl}(\boldsymbol{v}_k)$, we use subgradients instead.

Protrusion of an Object from a Cuboid Container. Consider a cuboid container $C = [l_x, u_x] \times [l_y, u_y] \times [l_z, u_z]$. Assume that C is large enough to contain any single sphere S_{ij} and sphere $S_{ij}(\boldsymbol{v}_i)$ protrudes from C. Let $(x_{ij}, y_{ij}, z_{ij})^{\mathsf{T}} = \boldsymbol{c}_{ij}(\boldsymbol{v}_i)$ and $\boldsymbol{p} = (p_x, p_y, p_z)^{\mathsf{T}}$ where

$$p_x = \begin{cases} x_{ij} - r_{ij} - l_x & \text{if } x_{ij} < l_x + r_{ij}, \\ 0 \quad \text{if } l_x + r_{ij} \leq x_{ij} \leq u_x - r_{ij}, \\ x_{ij} + r_{ij} - u_x & \text{if } u_x - r_{ij} < x_{ij}, \end{cases} \tag{3.2}$$

and we define p_y and p_z in the same way. By the assumption that C is enough large to contain any single sphere S_{ij}, exactly one of the three cases (3.2) is always satisfied. Then, the penetration depth is calculated by $\delta(S_{ij}(\boldsymbol{v}_i), \text{cl}(\overline{C})) = \|\boldsymbol{p}\|$, and $f_{ij}^{\text{pro}}(\boldsymbol{v})$ is differentiable as follows.

$$\begin{aligned} \frac{\partial f_{ij}^{\text{pro}}(\boldsymbol{v})}{\partial \boldsymbol{v}_a} &= \boldsymbol{0}, \quad \text{if } a \in \{1, \ldots, m\} \setminus \{i\}, \\ \frac{\partial f_{ij}^{\text{pro}}(\boldsymbol{v})}{\partial \boldsymbol{v}_i} &= 2 \cdot \frac{\partial \boldsymbol{c}_{ij}(\boldsymbol{v}_i)^{\mathsf{T}}}{\partial \boldsymbol{v}_i} \cdot \boldsymbol{p}. \end{aligned}$$

Motions of Objects

In multi-sphere scheme, we can impose a restriction on the motion of each object O_i by defining its motion function Λ_i appropriately. In this chapter, we show three different motions of objects for examples.

Translation. First we consider the case where each object O_i can translate by a vector $\boldsymbol{v}_i = (x_i, y_i, z_i)^\mathsf{T}$ in $\mathbb{R}^3$, but not allowed to rotate. For $\boldsymbol{x} \in \mathbb{R}^3$, we let

$$\Lambda_i(\boldsymbol{x}, \boldsymbol{v}_i) = \boldsymbol{x} + \boldsymbol{v}_i.$$

Then it holds

$$\frac{\partial \boldsymbol{c}_{ij}(\boldsymbol{v}_i)}{\partial \boldsymbol{v}_i} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Translation with a Fixed Direction. We next consider the case where object O_i is allowed to translate only in a prescribed direction in $\mathbb{R}^3$, but not allowed to rotate. Assume that the reference point $\boldsymbol{r}_i$ of object O_i lies on a line $\boldsymbol{d}_i + t_i \boldsymbol{e}_i$, where $\boldsymbol{d}_i, \boldsymbol{e}_i \in \mathbb{R}^3$ are given and t_i is a variable. Then we have

$$\begin{aligned} \Lambda_i(\boldsymbol{x}, t_i) &= \boldsymbol{x} - \boldsymbol{r}_i + \boldsymbol{d}_i + t_i \boldsymbol{e}_i, \\ \frac{\partial \boldsymbol{c}_{ij}(t_i)}{\partial t_i} &= \boldsymbol{e}_i. \end{aligned}$$

Translation and Rotation. Finally, we consider the case where each object O_i in $\mathbb{R}^3$ is allowed to translate and rotate around its reference point $\boldsymbol{r}_i$. Let $(x_i, y_i, z_i)^\mathsf{T}$ be the translation vector, $(\phi_i, \theta_i, \psi_i)$ be the z-x-z Euler angles, and $R_3(\phi_i, \theta_i, \psi_i)$ be the rotation matrix. Given variables $\boldsymbol{v}_i = (x_i, y_i, z_i, \phi_i, \theta_i, \psi_i)^\mathsf{T}$, we define the resulting position of a point $\boldsymbol{x} \in \mathbb{R}^3$ after the motion by

$$\Lambda_i(\boldsymbol{x}, \boldsymbol{v}_i) = R_3(\alpha_{\mathrm{rot}}\phi_i, \alpha_{\mathrm{rot}}\theta_i, \alpha_{\mathrm{rot}}\psi_i)(\boldsymbol{x} - \boldsymbol{r}_i) + (x_i, y_i, z_i)^\mathsf{T} + \boldsymbol{r}_i,$$

where a positive parameter α_{rot} denotes sensitivity of rotations. Then we have

$$\begin{aligned} \frac{\partial \boldsymbol{c}_{ij}(\boldsymbol{v}_i)}{\partial x_i} &= (1, 0, 0)^\mathsf{T}, \\ \frac{\partial \boldsymbol{c}_{ij}(\boldsymbol{v}_i)}{\partial \phi_i} &= \frac{\partial R_3(\alpha_{\mathrm{rot}}\phi_i, \alpha_{\mathrm{rot}}\theta_i, \alpha_{\mathrm{rot}}\psi_i)}{\partial \phi_i}(\boldsymbol{c}_{ij} - \boldsymbol{r}_i). \end{aligned}$$

The other derivatives of $\boldsymbol{c}_{ij}(\boldsymbol{v}_i)$ with respective to y_i, z_i, θ_i, and ψ_i can be calculated analogously.

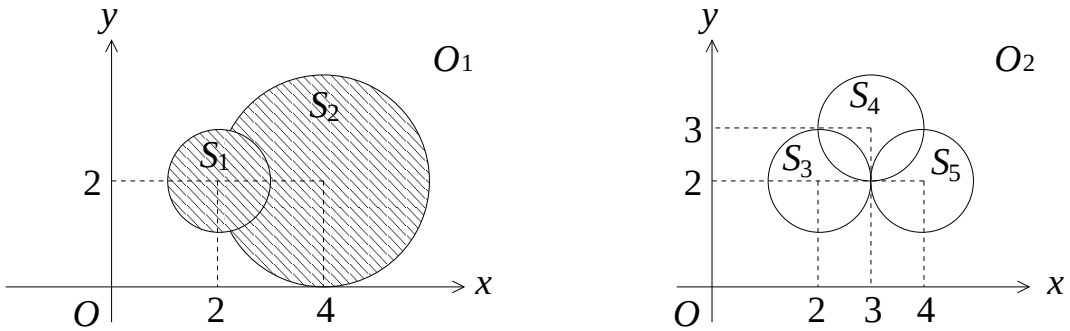


Figure 3.2: Two objects of a sample instance

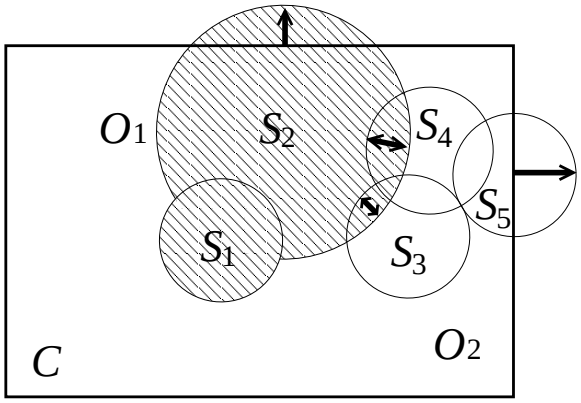


Figure 3.3: A layout of the two objects in Figure 3.2

Example with Translation and Rotation

We show an example of computation of the objective function and the gradient with an instance for a two-dimensional problem, where translations and rotations of objects are allowed. For simplicity, we omit indeces of objects added to the penalty functions and spheres in the previous section. Let $C = [0, 7] \times [0, 5]$ be a container and $O_1 = \{S_1, S_2\}$ and $O_2 = \{S_3, S_4, S_5\}$ are objects to be placed (see Figure 3.2), where $S_1, \ldots, S_5$ are spheres whose centers are $(2, 2), (4, 2), (2, 2), (3, 3)$ and $(4, 2)$, respectively, and the radii are 1, 2, 1, 1 and 1, respectively. We also choose a point $(2, 2)$ as the reference point for both objects and let $w_{\mathrm{pen}} = w_{\mathrm{pro}} = 1$ and $\alpha_{\mathrm{rot}} = 10^{-1}$. Let $\boldsymbol{v}_1 = (x_1, y_1, \theta_1)$, $\boldsymbol{v}_2 = (x_2, y_2, \theta_2)$ be the variables for O_1 and O_2, respectively and $\boldsymbol{v} = \{x_1, y_1, \theta_1, x_2, y_2, \theta_2\}$ represent the whole variables.

We consider a layout where $\boldsymbol{v}_1 = (0, 0, 20\pi/3)$ and $\boldsymbol{v}_2 = (3, 0, 10\pi/3)$ (see Figure 3.3). The centers of spheres $S_1, \ldots, S_5$ move to $(2, 2), (3, 2 + \sqrt{3}), (5, 2), (5 + (\sqrt{3} - 1)/2, 2 + (\sqrt{3} + 1)/2)$ and $(5 + \sqrt{3}, 3)$, respectively. The penalties $f_{23}^{\mathrm{pen}}(\boldsymbol{v}), f_{24}^{\mathrm{pen}}(\boldsymbol{v}), f_{2}^{\mathrm{pro}}(\boldsymbol{v})$ and $f_{5}^{\mathrm{pro}}(\boldsymbol{v})$ are not zero

(bold arrows in Figure 3.3). The values of the penalties are

$$f_{23}^{\mathrm{pen}}(\boldsymbol{v}) \approx 0.125,$$
$$f_{24}^{\mathrm{pen}}(\boldsymbol{v}) \approx 0.367,$$
$$f_{2}^{\mathrm{pro}}(\boldsymbol{v}) \approx 0.536,$$
$$f_{5}^{\mathrm{pro}}(\boldsymbol{v}) \approx 0.536,$$

and their gradients are

$$\nabla f_{23}^{\mathrm{pen}}(\boldsymbol{v}) \approx (0.536, -0.464, -0.139, -0.536, 0.464, 0.000),$$
$$\nabla f_{24}^{\mathrm{pen}}(\boldsymbol{v}) \approx (1.20, -0.185, -0.226, -1.197, 0.185, 0.170),$$
$$\nabla f_{2}^{\mathrm{pro}}(\boldsymbol{v}) \approx (0.000, 1.46, 0.146, 0.000, 0.000, 0.000),$$
$$\nabla f_{5}^{\mathrm{pro}}(\boldsymbol{v}) \approx (0.000, 0.000, 0.000, 1.46, 0.000, -0.146).$$

Hence the total penalties $F_{\mathrm{rigid}}(\boldsymbol{v})$ and its gradient $\nabla F_{\mathrm{rigid}}(\boldsymbol{v})$ are

$$F_{\mathrm{rigid}}(\boldsymbol{v}) \approx 1.56,$$
$$\nabla F_{\mathrm{rigid}}(\boldsymbol{v}) \approx (1.73, 0.815, \underline{-0.219}, -0.269, 0.649, \underline{0.0239}). \tag{3.3}$$

If we change the sensitivity α_{rot} of rotations from 0.1 to 1 and modify the initial variables $\boldsymbol{v}_1' = (0, 0, 2\pi/3)$ and $\boldsymbol{v}_2' = (3, 0, \pi/3)$ so that the layout does not change, then we have different gradient values

$$\nabla F_{\mathrm{rigid}}(\{\boldsymbol{v}_1', \boldsymbol{v}_2'\}) \approx (1.73, 0.815, \underline{-2.19}, -0.269, 0.649, \underline{0.239}). \tag{3.4}$$

For $\alpha_{\mathrm{rot}} = 10$, $\boldsymbol{v}_1'' = (0, 0, 2\pi/30)$ and $\boldsymbol{v}_2'' = (3, 0, \pi/30)$ it holds

$$\nabla F_{\mathrm{rigid}}(\{\boldsymbol{v}_1'', \boldsymbol{v}_2''\}) \approx (1.73, 0.815, \underline{-21.9}, -0.269, 0.649, \underline{2.39}). \tag{3.5}$$

Note that the layout with $\alpha_{\mathrm{rot}} = 10$, $\boldsymbol{v}_1''$ and $\boldsymbol{v}_2''$ is also same as Figure 3.3.

The underlined numbers in (3.3), (3.4) and (3.5) show that the derivatives with respective to the rotations differ depending on the sensitivity α_{rot} of the rotations.

3.2.2 Local Search Algorithm

This section describes our local search algorithm RIGIDQN for all motions of objects. Given a layout $(\mathcal{O}, C)$ as an initial solution, our local search algorithm RIGIDQN$(\mathcal{O}, C)$ returns a locally optimal solution computed by solving the penalized rigid sphere set packing problem (3.1) with quasi-Newton method. RIGIDQN moves the objects simultaneously and modifies the entire layout gradually until the value of the penalty function F_{rigid} becomes 0. In

practice, RIGIDQN is implemented so that the computation halts if the value of $F_{\rm rigid}$ becomes sufficiently small. In our experiments, we consider the case where the objects are allowed to translate and rotate in our experiments. RIGIDQN has three parameters $w_{\rm pen}$, $w_{\rm pro}$, and $\alpha_{\rm rot}$: $w_{\rm pen}$ (resp., $w_{\rm pro}$) denotes the weights of the penetration penalty (resp., the protrusion penalty), and $\alpha_{\rm rot}$ denotes the sensitivity of rotations. Since we observe from preliminary experiments that if $\alpha_{\rm rot}$ is not chosen appropriately, the local search algorithm does not work well from preliminary experiments. In general, a large $\alpha_{\rm rot}$ tends to rotate the corresponding object by an unnecessarily large amount. We let $w_{\rm pen} = w_{\rm pro} = 1$ and $\alpha_{\rm rot} = 10^{-2}$ in our experiments.

3.2.3 Iterated Local Search Algorithm

This section describes our iterated local search algorithm ILS_RIGID for the case where objects are allowed to translate and rotate. ILS_RIGID$(\mathcal{O}, C)$ generates an initial solution by placing objects $\mathcal{O}$ randomly so that the reference points are in the container C, rotates the objects randomly and applies RIGIDQN to the initial layout. ILS_RIGID maintains the incumbent solution $\mathcal{O}_{\rm opt}$ that minimizes the objective function $F_{\rm rigid}$ of (3.1), which will be used for generating the next initial solution. ILS_RIGID first perturbs the incumbent solution by swapping two randomly chosen objects O_1 and O_2 and rotates them randomly. Then, ILS_RIGID translates and rotates objects by invoking RIGIDQN. If the value of $F_{\rm rigid}$ for a new locally optimal solution is less than that of the incumbent solution, we update the incumbent solution with the locally optimal solution. ILS_RIGID repeats these operations until a time limit. ILS_RIGID is described formally in Algorithm 6.

Algorithm 6 : ILS_RIGID$(\mathcal{O}, C)$

```
Generate an initial layout O_opt of objects O by placing them randomly;
while within a time limit do
  Let O' := O_opt;
  Choose two objects O_1 and O_2 randomly from O';
  Swap O_1 and O_2 by their reference points;
  Rotate O_1 and O_2 randomly;
  Let O' := RIGIDQN(O', C);
  if F_rigid(O') < F_rigid(O_opt) then
    Let O_opt := O'
  end if
end while;
Return O_opt.
```

3.3 Collision Detection of Spheres

3.3.1 Motivation

In the multi-sphere scheme, we iteratively detect the collisions of spheres in different layouts. Preliminary experiments revealed that this procedure was the bottleneck of RigidQN and that the layout of spheres after each iteration may change drastically. Thus, we need a collision detection algorithm of spheres that does not exploit the information on the previous layouts.

In this section, we present an algorithm based on slab partitioning technique and a plane sweep method. We show that our algorithm requires $O(n \log n + K)$ time and $O(n + K)$ space if the dimension d and the maximum ratio ρ of radii of two spheres are constant, where n is the number of spheres and K is the number of pairs of colliding spheres. To explain our method, we deal with open spheres (the case for closed spheres can be treated analogously).

3.3.2 Related Work

The following property implies that $\Omega(n \log n + K)$ is a lower bound on the time complexity of any algorithm that enumerates all pairs of colliding spheres in a given set of n open spheres even if all spheres are on the same straight line and the radius of each sphere is 1, where K is the number of pairs of colliding spheres.

Lemma 2 ([85], p.334, Corollary 8.1.) *In the algebraic decision-tree model, any algorithm that determines whether a given set of n numbers contains some two numbers that differ from each other by less than $\varepsilon > 0$ requires $O(n \log n)$ test.* □

Collision detection of spheres has been extensively studied. Hopcroft et al. [51] presented an algorithm for detecting whether a given set of n spheres in $\mathbb{R}^3$ contains two intersecting spheres or not, which runs in $O(n \log^2 n)$ time and $O(n \log n)$ space. Kim et al. [64] considered a problem that asks to find the inclusion hierarchy among n spheres in $\mathbb{R}^2$, assuming that any two spheres has no intersection on their circumferences. They proposed a plane sweep algorithm, which requires $O(n \log n)$ time. Leszczynski and Ciesielski [68] proposed a simple plane sweep algorithm to detect all pairs of colliding spheres in $\mathbb{R}^2$ and $\mathbb{R}^3$, and showed that the algorithm performed well in practice.

Spatial subdivision is a basic approach of collision detection in computational geometry, e.g., the point location problem [82]. It is also used in collision detection [63, 91]. However, to the best of our knowledge, theoretical analysis of the performance is not known. Other approaches are also studied, e.g., hierarchies of bounding spheres [3, 55, 83]; see a survey by Lin and Gottschalk [71] and a textbook by Ericson [34].

In the field of simulation of moving spheres (particles, granules), collision detection are executed iteratively and the layouts after each iteration vary slightly. Data structures designed for this characteristic have been proposed. Ferrez [35] proposed a framework of physical simulation of a set of spheres in $\mathbb{R}^3$ and a fast algorithm to detect the collisions of spheres based on the power diagram [13, 59]. Agarwal et al. [3] proposed the *deformable necklace* model, a flexible chain of spheres in which only adjacent spheres may intersect. They studied hierarchical bounding volume for collision and self collision detection using the power diagram. Gavrilova and Ronke [40] compared the power diagram, the regular spatial subdivision, the regular spatial tree, and the segment tree from the view points of theoretical analyses and computational experiments.

3.3.3 Algorithm

Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a set of d-dimensional spheres, $\boldsymbol{c}_i \in \mathbb{R}^d$ and $r_i \in \mathbb{R}$ be the center and radius of S_i, respectively. We assume that all spheres are open, i.e., two spheres S_i and S_j collide if and only if $\|\boldsymbol{c}_i - \boldsymbol{c}_j\| < r_i + r_j$, where $\|\cdot\|$ denotes the Euclidean norm. Note that it takes $O(d)$ time to check the collision. Let $r_{\min} = \min_{i=1,\ldots,n}\{r_i\}$, $r_{\max} = \max_{i=1,\ldots,n}\{r_i\}$, $^\mathsf{T}$ denote the transpose of a vector/matrix, $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_d$ be the unit vectors in $\mathbb{R}^d$, and $x_i = \boldsymbol{c}_i{}^\mathsf{T}\boldsymbol{e}_1$ be the first coordinate value of $\boldsymbol{c}_i$.

We propose an algorithm called ColDetect. It divides a bounding box B of $\mathcal{S}$ into slabs with same size, assigns each S_i in $\mathcal{S}$ to the slabs intersecting S_i, and applies a plane sweep method to collision detection of spheres for each slab. We explain the detail of the slab partitioning technique and the plane sweep method in the following subsections. ColDetect is formally described in Algorithm 7.

3.3.4 Slab Partitioning

In this section, we explain the slab partitioning of a bounding box of all spheres. Let

$$B = [\underline{b}_1, \overline{b}_1] \times \cdots \times [\underline{b}_d, \overline{b}_d]$$

be the minimal axis-aligned bounding box of $\mathcal{S}$, where $\underline{b}_i, \overline{b}_i \in \mathbb{R}$ and $\underline{b}_i < \overline{b}_i$ for $1 \leq i \leq d$. Also let $h = 2r_{\max}, m_i = \lfloor(\overline{b}_i - \underline{b}_i)/h\rfloor$ and $h_i = (\overline{b}_i - \underline{b}_i)/m_i$ for $i = 2, \ldots, d$. It clearly holds that $\overline{b}_i - \underline{b}_i \geq 2r_{\max}$ for all $i = 1, \ldots, d$ and $h \leq h_i < 2h$ for all $i = 2, \ldots, d$.

We split B into slabs

$$B(\xi_2, \ldots, \xi_d) = [\underline{b}_1, \overline{b}_1] \times H_{2\xi_2} \times \cdots \times H_{d\xi_d},$$
$$\xi_i = 1, \ldots, m_i, \quad i = 2, \ldots, d,$$

Algorithm 7 : ColDetect($\mathcal{S}$)

$Q := \emptyset$;
Sort $\mathcal{S} = \{S_1, \ldots, S_n\}$ in the increasing order of $(x_i - r_i)$;
Compute the minimal axis-aligned bounding box B that contains all spheres in $\mathcal{S}$;
/* $B = [\underline{b}_1, \overline{b}_1] \times \cdots \times [\underline{b}_d, \overline{b}_d]$ */
$h := 2r_{\max}$; $h_1 := \overline{b}_1 - \underline{b}_1$;
$m_i := \lfloor (\overline{b}_i - \underline{b}_i)/h \rfloor$, $h_i := (\overline{b}_i - \underline{b}_i)/m_i$, for $i = 2, \ldots, d$;
$T := \emptyset$;
/* T is a balanced binary tree. Let $T(\xi_2, \ldots, \xi_d)$ be a linked list of spheres associated with the key $(\xi_2, \ldots, \xi_d)$. */
for $i = 1$ to n **do**
  for $k = 2$ to d **do**
    $H_{kj} := [\underline{b}_k + (j-1)h_k, \underline{b}_k + jh_k]$, $j = 1, \ldots, m_k$;
    $\overline{\sigma}_k(i) := \max\{j \mid \boldsymbol{c}_i{}^{\mathsf{T}} \boldsymbol{e}_k + r_i \in H_{kj}, j = 1, \ldots, m_k\}$;
    $\underline{\sigma}_k(i) := \min\{j \mid \boldsymbol{c}_i{}^{\mathsf{T}} \boldsymbol{e}_k - r_i \in H_{kj}, j = 1, \ldots, m_k\}$
  end for;
  for all slabs $B(\xi_2, \ldots, \xi_d)$ with $\xi_k \in [\underline{\sigma}_k(i), \overline{\sigma}_k(i)]$, for $k = 2, \ldots, d$ **do**
    if T does not contain key $(\xi_2, \ldots, \xi_d)$ **then**
      Insert an empty linked list to $T(\xi_2, \ldots, \xi_d)$
    end if;
    Append S_i to linked list $T(\xi_2, \ldots, \xi_d)$
  end for
end for;
$\underline{\sigma} := \{\underline{\sigma}_2, \ldots, \underline{\sigma}_d\}$;
for each nonempty linked list $T(\xi_2, \ldots, \xi_d)$ **do**
  $Q := Q \cup$ PlaneSweep$(T(\xi_2, \ldots, \xi_d), \underline{\sigma}, B(\xi_2, \ldots, \xi_d))$
end for;
Return Q.

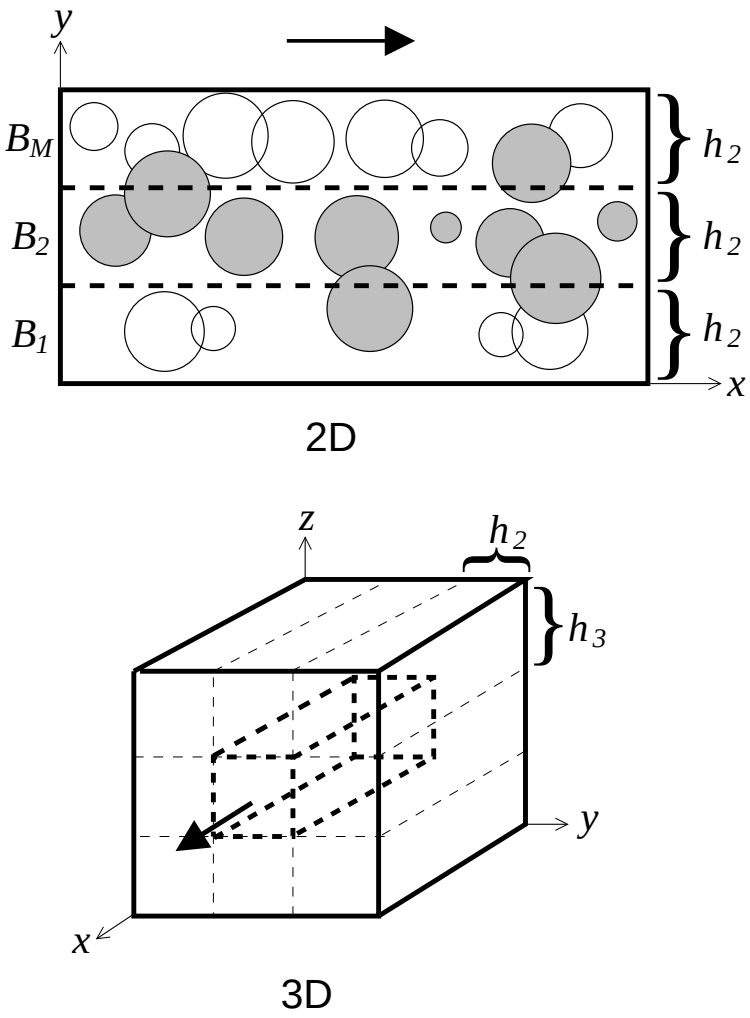


Figure 3.4: Examples of slab partitioning for 2D and 3D. Grey spheres are assigned to slab B_2 in 2D. Arrows represent the direction in which the plane sweep method sweeps.

where

$$H_{kj} = [\underline{b}_k + (j-1)h_k, \underline{b}_k + jh_k].$$

Note that each slab has size $(\overline{b}_1 - \underline{b}_1) \times h_2 \times \cdots \times h_d$. See Figure 3.4; "2D" and "3D" illustrate slab partitioning in $\mathbb{R}^2$ and $\mathbb{R}^3$, respectively, where the arrows represent the directions in which the plane sweep method sweeps spheres.

After splitting the bounding box into slabs, we assign each sphere to slabs that intersect the sphere. Any pair of colliding spheres is assigned to at least one slab. If at least one sphere is assigned to a slab, then we call the slab *nonempty*. Since $h_i \geq h = 2r_{\max}$, $i = 2, \ldots, d$, any sphere is assigned to up to two adjacent slabs in each dimension, i.e., any sphere is assigned to up to 2^{d-1} slabs in total.

Then we find pairs of colliding spheres for each slab and afterward report the union of the pairs as a solution. Some pairs of colliding spheres may be assigned to more than one slab whereas the solution should not include any same pair more than once. For each pair of colliding spheres, we determine one slab in which the pair will be detected, while we ignore the pair in the other slabs. Let

$$\sigma_k(i) = \{j \mid I_{ki} \cap H_{kj} \neq \emptyset,\ j = 1, \ldots, m_k\},$$
$$\underline{\sigma}_k(i) = \min \sigma_k(i), \quad \overline{\sigma}_k(i) = \max \sigma_k(i),$$

where

$$I_{ki} = [\boldsymbol{e}_k{}^{\mathsf{T}} \boldsymbol{c}_i - r_i,\ \boldsymbol{e}_k{}^{\mathsf{T}} \boldsymbol{c}_i + r_i].$$

Note that a sphere S_i is assigned to slabs $B(\xi_2, \ldots, \xi_d)$, $\xi_k \in \sigma_k(i)$, $k = 2, \ldots, d$. Since spheres are connected, it holds that $\sigma_k(i) = [\underline{\sigma}_k(i), \overline{\sigma}_k(i)]$. Assume that two spheres S_i and S_j collide each other. Then, (S_i, S_j) can be detected in slabs $B(\xi_2, \ldots, \xi_d)$ such that $\xi_k \in \sigma_k(i) \cap \sigma_k(j)$, for all $k = 2, \ldots, d$. We detect (S_i, S_j) only in a slab $B(\xi'_2, \ldots, \xi'_d)$ with $\xi'_k = \max\{\underline{\sigma}_k(i), \underline{\sigma}_k(j)\}, k = 2, \ldots, d$ and ignore the pair in the other slabs.

3.3.5 Plane Sweep Method

We exploit a simple plane sweep algorithm PlaneSweep for collision detection of spheres assigned to a slab. The idea is derived from a method by Leszczynski and Ciesielski [68]. We explain the case where we sweep spheres from left to right along $\boldsymbol{e}_i$ without loss of generality.

Assume that we are given spheres $\mathcal{S} = \{S_1, \ldots, S_n\}$, which are indexed in the increasing order of $(x_i - r_i)$. For each sphere S_i, we check a collision of S_i with S_j such that

$$i < j \leq n \quad \text{and} \quad x_i + r_i > x_j - r_j.$$

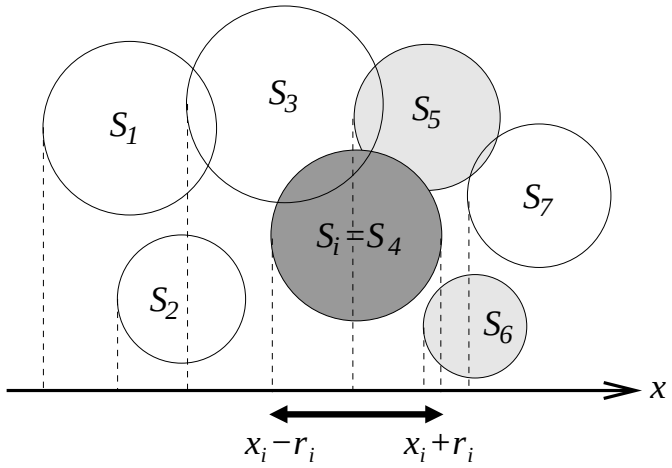


Figure 3.5: An illustration of plane sweep method; To check a collision of $S_i = S_4$ with S_j, the method picks up S_5 and S_6.

See Figure 3.5, where PlaneSweep picks up S_5 and S_6 to check collisions between S_4 and them. PlaneSweep is formally described in Algorithm 8.

Since ColDetect first sorts the entire spheres in the increasing order of $(x_i - r_i)$ and assigns them to slabs one by one in the order, PlaneSweep does not need to sort the spheres given by ColDetect.

Algorithm 8 : PlaneSweep($\mathcal{S}, \underline{\sigma}, B'$)

```
/* Assume that B' = B(ξ_2, ..., ξ_d). */
/* Spheres S = {S_1, ..., S_n} are indexed in the increasing order of (x_i − r_i). */
Q := ∅;  i := 1;
while i ≤ n do
  j := i + 1;
  while j ≤ n and x_i + r_i > x_j − r_j do
    if S_i and S_j collide each other and ξ_k = max{σ_k(i), σ_k(j)}, for all k = 2..., d then
      Q := Q ∪ {(S_i, S_j)}
    end if;
    j := j + 1
  end while;
  i := i + 1
end while;
Return Q.
```

3.3.6 Performance Analysis

In this section, we analyze the performance of ColDetect. We first show the time and space complexity of the plane sweep algorithm PlaneSweep for a slab. We consider a graph

whose nodes correspond to the spheres and edges correspond to the pairs of spheres checked by PlaneSweep, and prove an upper bound of the number of the edges of the graph. Finally, we prove that our algorithm ColDetect runs in $O(n \log n + K)$ time and $O(n + K)$ space if the dimension d and the maximum ratio $\rho = r_{\max}/r_{\min}$ of radii of two spheres are constant, where n denotes the number of spheres and K denotes the number of pairs of colliding spheres.

Property of a Graph

Let $G = (V, E \cup A)$ be a simple undirected graph with a vertex set V and an edge set $E \cup A$ such that $E \cap A = \emptyset$.

For a vertex v, let $E(v)$ (resp., $A(v)$) denote the set of edges in E (resp., A) that are incident to v, and $N(v)$ denote the neighbour of a vertex v in G. In particular, let $N_E(v)$ (resp., $N_A(v)$) denote the set of vertices adjacent to v via edges in E (resp., A).

Let γ_v be the maximum size of an independent set $I \subseteq N_A(v)$ in graph (V, E), and let $\gamma = \max_{v \in V} \gamma_v$.

Theorem 3 $|A| \leq \gamma|V| + 2\gamma|E|$. □

To prove Theorem 3, we assign each edge $a \in A$ to some vertex $v \in V$ or some edge $e \in E$. Let $M(v)$ (resp., $M(e)$) denote the set of edges $a \in A$ that are assigned to a vertex $v \in V$ (resp., edge $e \in E$). It suffices to show that there exists an assignment such that $|M(v)| \leq \gamma$, $v \in V$ and $|M(e)| \leq 2\gamma$, $e \in E$. We show that such an assignment can be constructed by the next procedure.

$M(u) := M(e) := \emptyset$ for all $u \in V$ and $e \in E$;
for all $x \in V$ **do**
  $Y := N_A(x)$;
  while there exists a vertex $y \in Y$ with $|N_E(y) \cap Y| \leq |N_E(x)|$ **do**
    Choose such a vertex y;
    $M(x) := M(x) \cup \{(x, y)\}$;
    We denote $N_E(y) \cap Y$ by $\{y_i \mid i = 1, 2, \ldots, |N_E(y) \cap Y|\}$;
    Choose $|N_E(y) \cap Y|$ edges $\{e_i \mid i = 1, 2, \ldots, |N_E(y) \cap Y|\} \subseteq E(x)$;
    $M(e_i) := M(e_i) \cup \{(x, y_i)\}$ for $i = 1, 2, \ldots, |N_E(y) \cap Y|$;
    $Y := Y - (N_E(y) \cup \{y\})$
  end while
end for.

Lemma 4 *Each edge $a = (u, v) \in A$ is assigned to at least one of $M(u)$, $M(v)$, and $M(e)$ for some edge $e \in E(u) \cup E(v)$.*

Proof: Let $a = (u, v)$ be an arbitrary edge in A. Consider when u is chosen as x in the for-loop. If a is assigned to $M(x) = M(u)$ or $M(e)$ for some edge $e \in E(x) = E(u)$ in the while-loop, then we are done. Assume that the edge a is not assigned to any of $M(x) = M(u)$ and $M(e)$ with $e \in E(x) = E(u)$ in the while-loop for $x = u$. In this case, we claim that the edge a is assigned to $M(x) = M(v)$ or $M(e)$ for some edge $e \in E(x) = E(v)$ in the while-loop when v is chosen as x in the for-loop. Since $a = (u, v)$ is not assigned in the while-loop when $x = u$, this means that

$$|N_E(v) \cap Y| > |N_E(u)|$$

holds immediately after the while-loop for $x = u$ terminates. From this, we have

$$|N_E(u)| < |N_E(v)|,$$

which implies that when v is chosen as x in the for-loop, u cannot belong to the final Y after termination of the while-loop for $x = v$. That is, a is assigned to $M(v)$ or $M(e)$ for some edge $e \in E(v)$ when $x = v$, as claimed. □

Lemma 5 *It holds that $|M(v)| \leq \gamma_v$ for each $v \in V$ and $|M(e)| \leq \gamma_u + \gamma_v$ for each $e = (u, v) \in E$.*

Proof: For a fixed $x \in V$, the number of iterations in the while-loop is at most γ_x since the set of vertices chosen as y forms an independent set $I \subseteq N_A(x)$ in graph (V, E). Hence the number of edges $a \in A$ which $M(x)$ (resp., $M(e)$ with $e \in E(x)$) receives in the while-loop is at most γ_x. For each $e = (u, v) \in E$, $M(e)$ receives at most γ_u (resp., γ_v) edges in A when u (resp., v) is chosen as x in the for-loop. □

Theorem 3 follows from Lemmas 4 and 5.

Collision Detection for a Slab

We next show the time and space complexity of the plane sweep method for a slab.

Lemma 6 *Let W be a slab with size $h_1 \times \cdots \times h_d$ and $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a set of d-dimensional spheres that intersect W, where $\boldsymbol{c}_i$ and r_i denote the center and radius of S_i, respectively. Assume that $S_1, \ldots, S_n$ are indexed in the increasing order of $(\boldsymbol{c}_i{}^{\mathsf{T}}\boldsymbol{e}_1 - r_i)$, where $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_d$ are the unit vectors in $\mathbb{R}^d$. Also let $r_{min} = \min_{i=1,\ldots,n}\{r_i\}$, $r_{max} = \max_{i=1,\ldots,n}\{r_i\}$, $\rho = r_{max}/r_{min}$, $\mu = \max_{i=2,\ldots,d}\{h_i\}/r_{max}$, and K be the number of pairs of colliding spheres. If ρ, d, and μ are constant,* PlaneSweep *detects all pairs of colliding spheres in $O(n + K)$ time and $O(n + K)$ space.*

Proof: It is obvious that PLANESWEEP needs $O(n+K)$ space. Note that the time complexity of PLANESWEEP is dominated by the number of checked pairs of spheres because it takes $O(d)$ time to check a collision of two spheres and we assume that d is constant. Let

$$\begin{aligned}
V &= \{S_1, \ldots, S_n\}, \\
E &= \{(S_i, S_j) \mid S_i \text{ and } S_j \text{ collide}, 1 \le i < j \le n\}, \\
A &= \{(S_i, S_j) \mid S_i \text{ and } S_j \text{ do not collide}, \\
&\qquad\qquad \boldsymbol{c}_i{}^{\mathsf{T}} e_1 + r_i > \boldsymbol{c}_j{}^{\mathsf{T}} e_1 - r_j,\ 1 \le i < j \le n\}.
\end{aligned}$$

Note that $|V| = n$, $|E| = K$, and $E \cup A$ are the pairs of spheres checked in PLANESWEEP, and $E \cap A = \emptyset$. By Theorem 3, it holds that $|E| + |A| \le \gamma|V| + (2\gamma+1)|E|$. By the definition, $\gamma = \max_{S \in \mathcal{S}} \gamma_S$, where γ_S be the maximum size of an independent set $I \subseteq N_A(S)$ in a graph (V, E). It holds that γ_S is less than or equal to the size of the maximum subset $J \subseteq N_A(S)$ such that spheres in J do not collide each other.

Now we consider a minimal axis-aligned bounding box B_{AS} of the spheres $N_A(S)$. The edge length along $\boldsymbol{e}_1$ is up to $2r + 4r_{\max} \le 6r_{\max}$, where r denotes the radius of S, and the edge length along $\boldsymbol{e}_i$ is up to $h_i + 4r_{\max}$ for all $i = 2, \ldots, d$ (see Figure 3.6). Since $h_i \le \mu r_{\max}$ for $i = 2, \ldots, d$ by the assumption, the volume of B_{AS} is less than or equal to $6(\mu+4)^{d-1} r_{\max}^d$. Note that the volume of a sphere with a radius r is $\pi^{d/2} r^d / \Gamma(d/2+1)$, where Γ is the gamma function. Thus, it holds that

$$\begin{aligned}
\gamma_S &\le 6(\mu+4)^{d-1} r_{\max}^d \Big/ \frac{\pi^{\frac{d}{2}} r_{\min}^d}{\Gamma\left(\frac{d}{2}+1\right)} \\
&= 6\pi^{-\frac{d}{2}} \Gamma\left(\frac{d}{2}+1\right) \rho^d (\mu+4)^{d-1}.
\end{aligned}$$

Since this bound is independent of S, it also holds that

$$\gamma = \max_{S \in \mathcal{S}} \{\gamma_S\} \le 6\pi^{-\frac{d}{2}} \Gamma\left(\frac{d}{2}+1\right) \rho^d (\mu+4)^{d-1}.$$

Since ρ, d, and μ are constant by the assumption, it holds that $\gamma = O(1)$ and $|E| + |A| = O(n+K)$. Therefore, PLANESWEEP runs in $O(n+K)$ time. □

Collision Detection of All Spheres

Theorem 7 *Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a set of d-dimensional spheres with radii $r_1, \ldots, r_n$, $r_{min} = \min_{i=1,\ldots,n}\{r_i\}$, $r_{max} = \max_{i=1,\ldots,n}\{r_i\}$, and $\rho = r_{max}/r_{min}$. Algorithm* COLDETECT *correctly detects all pairs of colliding spheres and runs in $O(n \log n + K)$ time and $O(n+K)$ space if d and ρ are constant, where K denotes the number of pairs of colliding spheres.*

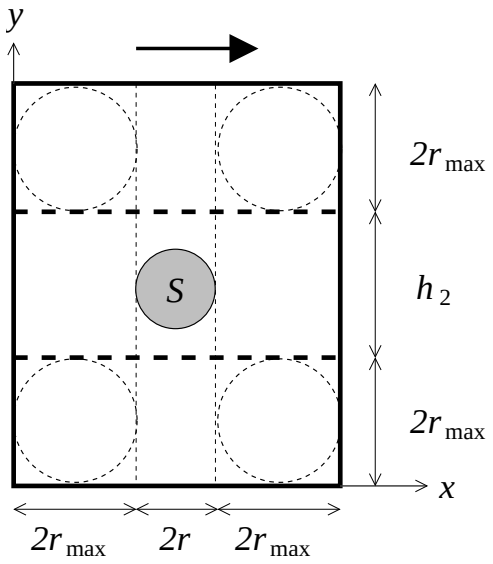


Figure 3.6: The minimal axis-aligned bounding box of $N_A(S)$ for 2D should be totally contained in the bounding box drawn in the bold lines. The arrow represents the direction in which PLANESWEEP sweeps spheres.

Proof: COLDETECT first sorts the spheres in $O(n \log n)$ time.

We then consider the slab partitioning. We store the spheres assigned to a nonempty slab $B(\xi_2, \ldots, \xi_d)$ in a linked list and store the pointer to the linked list in a balanced binary tree such as a red-black tree by associating a key $(\xi_2, \ldots, \xi_d)$ with the pointer, where the key is in the lexicographic order. Let L be the number of nonempty slabs. Since it takes $O(d)$ time to compare two keys, it takes $O(d \log L)$ time to assign a sphere to a slab. It takes $O(L)$ time to enumerate all nonempty slabs. Note that we do not need to see any empty slab to enumerate all nonempty slabs.

Let $\mathcal{B}$ be the set of nonempty slabs, $\mathcal{S}_Y$ be the spheres assigned to a nonempty slab Y, $n_Y = |\mathcal{S}_Y|$, K_Y be the number of pairs of colliding spheres in $\mathcal{S}_Y$, $\hat{n} = \sum_{Y \in \mathcal{B}} n_Y$, and $\hat{K} = \sum_{Y \in \mathcal{B}} K_Y$. Since each sphere is assigned to up to 2^{d-1} slabs, it holds that $n \leq \hat{n} \leq 2^{d-1} n$, $L \leq \hat{n} \leq 2^{d-1} n$, and $K \leq \hat{K} \leq 2^{d-1} K$. Thus, it takes $O(\hat{n} d \log L)$ time to assign all spheres to slabs.

We next consider the plane sweep method. Since we let $h = 2r_{\max}$ and $h_i = (\overline{b}_i - \underline{b}_i)/\lfloor (\overline{b}_i - \underline{b}_i)/h \rfloor$, it holds that $h_i < 2h = 4r_{\max}$ for $i = 2, \ldots, d$ and $\mu = \max_{i=2,\ldots,d}\{h_i\}/r_{\max}$ is constant. Thus, it takes $O(n_Y + K_Y)$ time to apply PLANESWEEP to a nonempty slab Y by Lemma 6 under the assumption that d and ρ are constant. It also takes $O(L)$ time to enumerate all nonempty slabs in the balanced binary tree. Hence, the time complexity to apply PLANESWEEP to all nonempty slabs is $O(\hat{n} + \hat{K} + L)$.

Therefore, ColDetect runs in $O(n \log n + \hat{n}d \log L + \hat{n} + \hat{K} + L) = O(n \log n + K)$ time in total by the assumption that d is constant.

Finally, we now consider the space complexity. We need $O(n + K)$ space to store all spheres and all pairs of colliding spheres, $O(L)$ space to store nonempty slabs in the balanced binary tree, and $O(\hat{n})$ space to store spheres in the linked lists of the nonempty slabs. Also it requires $O(n_Y + K_Y)$ space to apply PlaneSweep to a nonempty slab Y and $O(\hat{n} + \hat{K})$ space to apply PlaneSweep to all nonempty slabs. Thus, ColDetect needs $O(n + K)$ space by the assumption. □

3.3.7 Uniform Spatial Subdivision

Collision detection algorithm based on the uniform spatial subdivision is a common approach, e.g., it is introduced in [34]. Instead of slabs, it divides a bounding box B of spheres $\mathcal{S}$ into cells with size $h_1 \times \cdots \times h_d$, where $h_i = (\overline{b}_i - \underline{b}_i)/\lfloor (\overline{b}_i - \underline{b}_i)/h \rfloor$ for all $i = 1, \ldots, d$. We assign each sphere to cells that intersect the sphere as in ColDetect. Then, we check all pairs of spheres assigned in each cell by a brute force method. We can prove that if the dimension and the maximum ratio of radii of two spheres are constant, then the algorithm runs in $O(n \log n + K)$ time with $O(n + K)$ space analogously with Theorem 7.

3.3.8 Summary

We proposed algorithm ColDetect for detecting all pairs of colliding spheres that runs in $O(n \log n + K)$ time and $O(n + K)$ space if both the dimension and the maximum ratio of radii of two spheres are constant.

It is left for future work to compare the algorithm based on the slab partitioning technique and that based on the uniform spatial subdivision through computational experiments. Also it is interesting to generalize our algorithm for objects with arbitrary shapes.

3.4 Implementation of the Collision Detection Algorithm

3.4.1 Introduction

In the previous section, we proposed a fast collision detection algorithm of spheres. In this section we show a slightly modified version of the collision detection algorithm in order to incorporate it into an implementation of the multi-sphere scheme. The new collision detection algorithm ColDetect2 divides a bounding box containing all spheres into thin slabs, and applies a plane sweep method to each slab as in the previous section. In ColDetect2, we manage slabs by an array instead of a balanced binary tree for simplicity. Moreover, we exploit the characteristic that we do not need to detect the collisions of spheres that belong to the same object. Our experimental results reveal that the new algorithm overwhelms the previous method based on axis-aligned bounding boxes of objects. We here explain the algorithm for the case of $\mathbb{R}^2$ (the case of $\mathbb{R}^3$ can be discussed analogously). The case for $\mathbb{R}^3$ is apparent from that of $\mathbb{R}^2$.

Let $\mathcal{O} = \{O_1, \ldots, O_m\}$ be the set of all objects in a given layout, where each object consists of a set of circles, $\mathcal{S} = \{S_1, \ldots, S_N\}$ be the set of all circles in the layout, where $\boldsymbol{c}_i = (x_i, y_i)$ is the center of circle S_i, r_i is the radius of S_i, and $\mathrm{obj}(S_i)$ is the index of the object which S_i belong to. We assume that all circles are open in this section, i.e., two circles S_i and S_j collide if and only if $\|\boldsymbol{c}_i - \boldsymbol{c}_j\| < r_i + r_j$.

3.4.2 Axis-aligned Bounding Box Based Method

For comparison, we use an axis-aligned bounding boxes (AABB) based collision detection algorithm. First we compute a bounding boxes of each object (which is now approximated by a set of circles) and check the collisions of every two bounding boxes. Then, for each pair of objects whose bounding boxes collide each other, we check all pairs of circles belong to them except for the self collisions.

3.4.3 New Algorithm

We propose a new collision detection algorithm ColDetect2 for the multi-sphere scheme based on a plane sweep method, slab partitioning, and skips of self collisions. First, it divides a minimal bounding box B of all circles $\mathcal{S}$ into slabs with a same height. Then, it applies a plane sweep method to collision detection of circles in each slab. Additionally, it avoids checking some pairs of circles that belong to the same object. ColDetect2 is formally described in Algorithm 9.

The slab partitioning is almost same as that in Section 3.3.4. The only difference is the data structure to store the spheres that intersect slabs. Each slab has a list of spheres that

intersect the slab. In Section 3.3.4, the lists are stored in a binary balanced tree. But, we store them in an array in ColDetect2. This is because spheres are usually placed densely in solutions of the packing problems and most slabs are likely to be nonempty. Also the implementation with an array becomes simpler than that with a binary balanced tree.

Algorithm 9 : ColDetect2($\mathcal{O}, h$)

$Q := \emptyset$;
$\mathcal{S} := \bigcup_{j=1}^{m} \{S \in O_j \mid O_j \in \mathcal{O}\}$;
Sort $\mathcal{S}$ in the increasing order of $(x_i - r_i)$;
Compute a minimal bounding box B that contains all circles $\mathcal{S}$;
Let W and H be the width and the height of B, respectively;
$M := \lfloor H/h \rfloor$, $h' := H/M$;
Divide B into M slabs $\{B_1, \ldots, B_M\}$ with a height h';
$\underline{\sigma}(i) := \min\{j \mid S_i \cap B_j \neq \emptyset, j = 1, \ldots, M\}$ for all $i = 1, \ldots, n$;
for $i = 1$ to M **do**
  $\mathcal{S}' := \{S \in \mathcal{S} \mid S \cap B_i \neq \emptyset\}$;
  Apply the plane sweep method to obtain a set Q' of pairs of colliding circles in $\mathcal{S}'$;
  $Q := Q \cup Q'$
end for;
Return Q.

Skip of Self Collisions

The plane sweep method PlaneSweep2 in ColDetect2 is also almost same as that in Section 3.3.5. The different point is that we speed up it by skipping the checks of self collisions. Since we do not need to check self collisions, which are the collisions of the circles that belong to the same object, we try to skip them in the plane sweep method. We take advantage of the characteristic of the multi-sphere scheme that circles which belong to the same object may appear consecutively in the circle sequence given to the plane sweep method because circles that belong to the same object are usually situated closely each other and each slab is sufficiently thin by the slab partitioning.

Assume that we are given a set $\mathcal{S} = \{S_1, \ldots, S_n\}$ of circles indexed in the increasing order of $(x_i - r_i)$. Then we define the skip value $skip[i]$ of circle S_i by $skip[i] = \arg\min\{j \mid \mathrm{obj}(S_i) \neq \mathrm{obj}(S_{i+j}), 0 < j \leq n - i\} \cup \{n - i + 1\}$. If S_i and S_j belong to the same object, then circles C_k with $k \in \{i, j, \ldots, j + skip[j] - 1\}$ also belong to the same object. Thus, we next check the collision of S_i and $S_{j+skip[j]}$ in the plane sweep method. We compute the skip values *skip* before applying the plane sweep method. PlaneSweep2 is formally described in Algorithm 10.

Algorithm 10 : PLANESWEEP2($\mathcal{S}$, *skip*, $\underline{\sigma}$, b)

/* Circles in $\mathcal{S} = \{S_1, \ldots, S_n\}$ are indexed in the increasing order of the leftmost x-coordinates. */
/* *skip*[i] represents the index of the first sphere in $\mathcal{S}$ after S_i that belongs the different object with S_i */
$Q := \emptyset$;
$i := 1$;
while $i \leq n$ **do**
  $j := i + 1$;
  while $j \leq n$ and $x_i + r_i > x_j - r_j$ **do**
    if obj(S_i) = obj(S_j) **then**
      $j := j +$ *skip*[j];
    else
      if S_i and S_j collide each other and $b = \max\{\underline{\sigma}(i), \underline{\sigma}(j)\}$ **then**
        $Q := Q \cup \{(S_i, S_j)\}$;
      end if
      $j := j + 1$
    end if;
  end while;
  $i := i + 1$
end while;
Return Q.

3.4.4 Environment

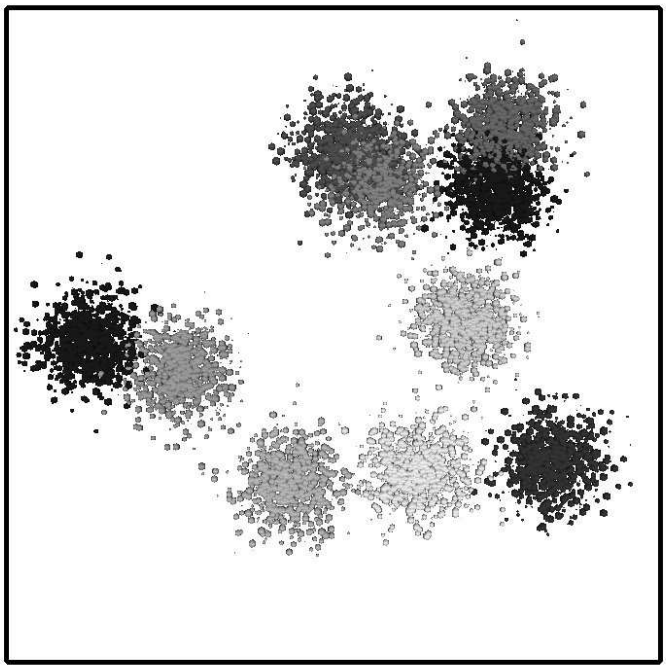
The following sections report the results on our computational experiments. We first show the performance of the new collision detection algorithm compared to that of the AABB based method. Then, we replace the collision detection algorithm in the iterated local search algorithm ILS_RIGID in Section 3.2.3 by the new one and compare the results.

We implemented our programs in C++, compiled it by GCC 4.0.2 and conducted experiments on a PC with an Intel Xeon 2.8GHz processor (NetBurst) and 1GB memory.

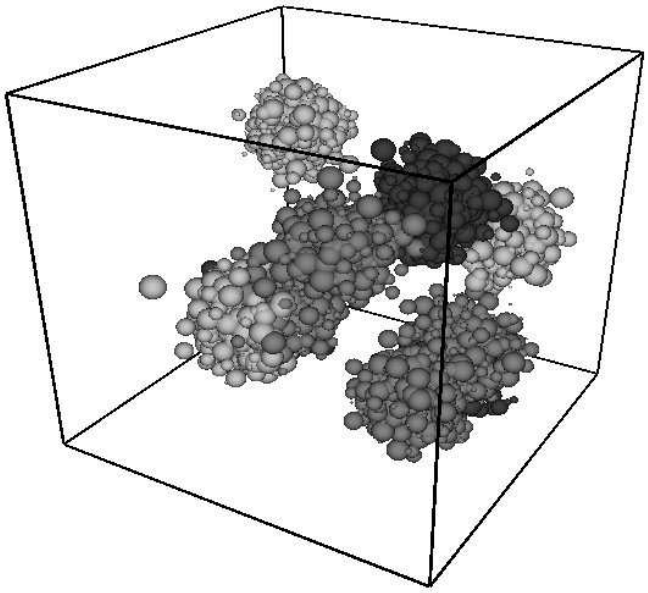
3.4.5 Experiments of Collision Detection

We compare computation times of an AABB based algorithm and our new algorithm COLDETECT2 for 2D and 3D. We use three different types of instances. First we fix the number of spheres per object at 1000 and conducted experiments varying the number of objects.

We generate instances as follows. Let m be the number of objects, n be the number of spheres per object, and d be the dimension. We set the radius of a circle by an uniform random number in $[1/10, 1]$ and set each coordinate of the center of the circle by a normal random number of $N(0, n^{2/d})$. We then place objects in a cube with edge length $(4mn)^{1/d}$ randomly. See Figure 3.7 for examples of 2D and 3D.



2D



3D

Figure 3.7: Examples of randomly generated layouts. There are 10 objects and each object consists of 1000 spheres in each layout.

Table 3.1: The average number of spheres and collisions with different numbers of objects (the number of spheres per object is 1000).

	2D		3D	
NO	ANS	ANC	ANS	ANC
40	4.0e+04	5.7e+04	4.0e+04	1.4e+05
100	1.0e+05	1.3e+05	1.0e+05	3.2e+05
200	2.0e+05	2.4e+05	2.0e+05	4.7e+05
400	4.0e+05	4.7e+05	4.0e+05	8.5e+05
600	6.0e+05	6.5e+05	6.0e+05	1.4e+06
800	8.0e+05	8.6e+05	8.0e+05	1.7e+06
1000	1.0e+06	1.1e+06	1.0e+06	2.3e+06
1200	1.2e+06	1.3e+06	1.2e+06	2.5e+06
1400	1.4e+06	1.5e+06	1.4e+06	3.0e+06

NO: the number of objects
ANS: the average number of spheres
ANC: the average number of collisions

Let "AABB" denote the AABB based method, "SWEEP" denote ColDetect2 with the slab partitioning and the plane sweep method without the skip of the self collisions, and "SWEEP+SKIP" denote ColDetect2 with the slab partitioning, the plane sweep method and the skip of the self collisions.

We fix the number of spheres per object at 1000 and vary slab height h over $\{\bar{r}, 2\bar{r}, 3\bar{r}, 4\bar{r}, 5\bar{r}\}$ and the number of objects from 10 to 1400, where $\bar{r}$ denotes the average radius of all spheres of an instance. We try 10 runs for each combination of the parameters and compare the average computation time of F_{pen}. Table 3.1 reports the average number of spheres and that of collisions.

We observe that "SWEEP" and "SWEEP+SKIP" outperforms "AABB" with any slab height h and both "SWEEP" and "SWEEP+SKIP" are slightly better for $h \in \{3\bar{r}, 4\bar{r}, 5\bar{r}\}$. Figure 3.8 shows the average computation time of 10 runs with $h = 4\bar{r}$. We plot two graphs for each dimension: the above one represents the graph of "SWEEP," "SWEEP+SKIP" and "AABB," and the below one represents the graph of "SWEEP" and "SWEEP+SKIP." "SWEEP+SKIP" runs faster than "SWEEP" by up to about 20%.

We then fix the number of objects at 10 and vary slab height h over $\{\bar{r}, 2\bar{r}, 3\bar{r}, 4\bar{r}, 5\bar{r}\}$ and the number of spheres per object from 500 to 10000. We generate layouts as in the first experiment, try 10 times for each combination of the parameters and compare the average computation time of F_{pen} as well. Table 3.2 reports the average number of spheres and that of collisions.

Table 3.2: The average number of spheres and collisions with different numbers of spheres per object (the number of objects is 10).

	2D		3D	
NSO	ANS	ANC	ANS	ANC
500	5.0e+03	6.2e+03	5.0e+03	2.7e+04
1000	1.0e+04	1.6e+04	1.0e+04	5.0e+04
2000	2.0e+04	2.4e+04	2.0e+04	9.7e+04
4000	4.0e+04	3.5e+04	4.0e+04	2.3e+05
6000	6.0e+04	6.2e+04	6.0e+04	1.9e+05
8000	8.0e+04	1.1e+05	8.0e+04	2.1e+05
10000	1.0e+05	1.3e+05	1.0e+05	2.8e+05

NSO: the number of spheres per object
ANS: the average number of spheres
ANC: the average number of collisions

We also observe that "SWEEP" and "SWEEP+SKIP" outperforms "AABB" with any slab height h and both "SWEEP" and "SWEEP+SKIP" are slightly better for $h \in \{3\bar{r}, 4\bar{r}, 5\bar{r}\}$. Figure 3.9 shows the average computation time of 10 runs with $h = 4\bar{r}$. In this experiment, "SWEEP+SKIP" runs faster than "SWEEP" by up to about 50%.

Table 3.3: The information of instances.

2D

Instance	TNO	TNS	ANS	Container
dighe1	16	1415	88.43	110 × 100
shapes0	43	3176	73.86	60 × 40
swim	48	5598	116.62	6500 × 5752

3D

Instance	TNO	TNS	ANS	Container
mol1	2	2575	1287.5	54×54×54
mol2	3	1948	649.3	46×46×46
mol3	3	2679	893.0	51×51×51

TNO: the total number of objects
TNS: the total number of spheres
ANS: the average number of spheres

Finally, we use the same instances as in [60] and conducted experiments. For each instance, we generate an initial layout by placing all objects randomly into the container with no protrusion of objects and compare the total computation time. Table 3.3 shows the in-

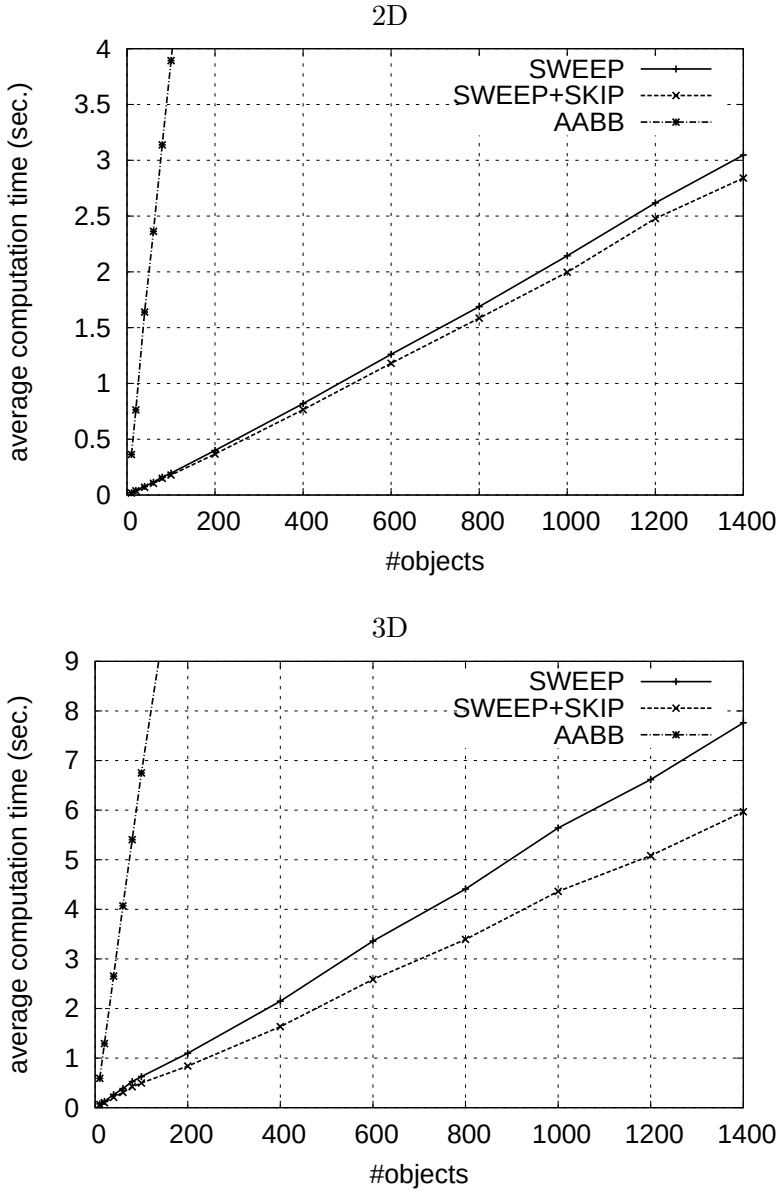


Figure 3.8: The average computation time of collision detection with different numbers of objects (the number of spheres per object is 1000).

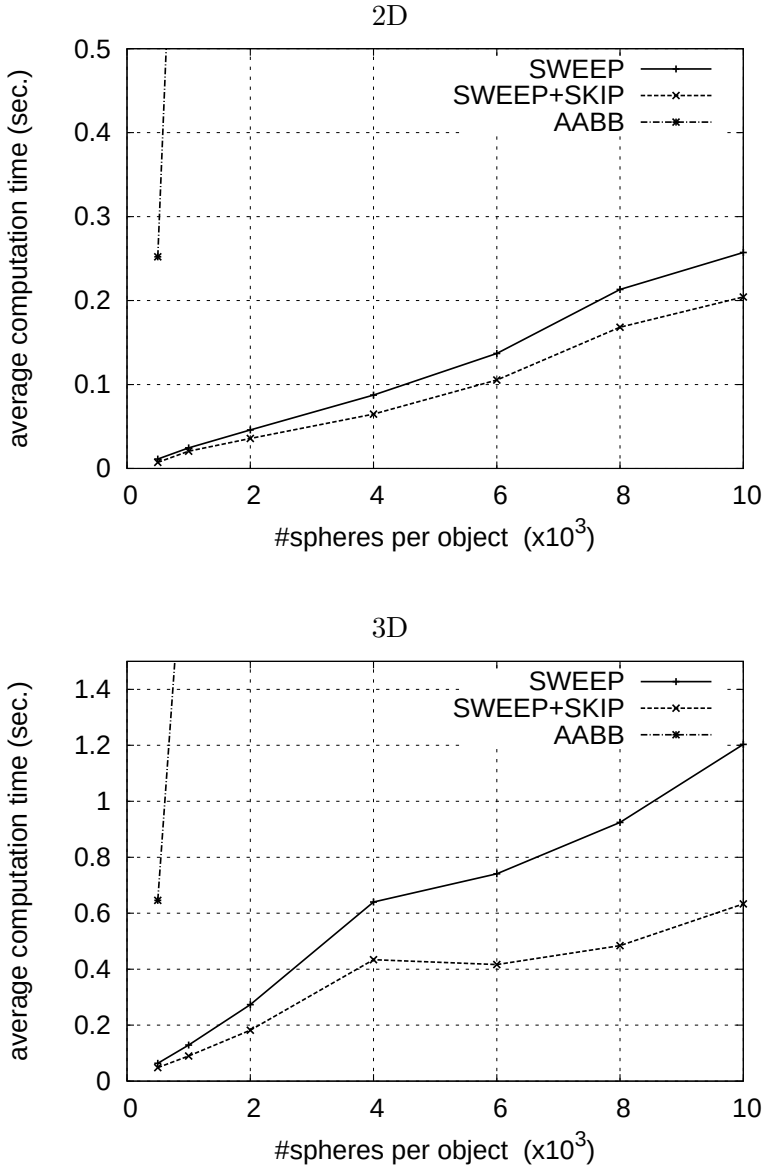


Figure 3.9: The average computation time of collision detection with different numbers of spheres per object (the number of object is 10).

Table 3.4: The total computation time of collision detection in seconds.

Instance	SWEEP					SWEEP + SKIP					AABB
	$\bar{r}$	$2\bar{r}$	$3\bar{r}$	$4\bar{r}$	$5\bar{r}$	$\bar{r}$	$2\bar{r}$	$3\bar{r}$	$4\bar{r}$	$5\bar{r}$	
dighe1	1.22	1.16	1.15	1.17	1.19	1.18	1.14	**1.13**	1.15	1.19	1.96
shapes0	1.20	1.08	1.09	1.09	1.13	1.11	1.09	**1.04**	1.06	1.08	3.15
swim	2.78	2.58	2.56	2.55	2.60	2.67	2.49	**2.47**	2.49	2.53	7.99
mol1	1.27	1.00	0.94	0.97	0.96	1.18	0.91	**0.86**	0.88	0.90	5.63
mol2	1.33	1.06	1.04	1.09	1.13	1.22	1.02	**0.99**	1.02	1.05	4.52
mol3	1.79	1.48	1.47	1.50	1.57	1.69	1.42	**1.37**	1.41	1.47	8.43

formation of the instances. Table 3.4 shows the total computation time in seconds of 100 runs. For "SWEEP" and "SWEEP+SKIP," we show the results with different h values. The fastest computation time of each instance is indicated in bold face. Table 3.4 also shows that "SWEEP" and "SWEEP+SKIP" run much faster than "AABB," while "SWEEP+SKIP" runs faster than "SWEEP." "SWEEP+SKIP" with $h = 3\bar{r}$ is the fastest.

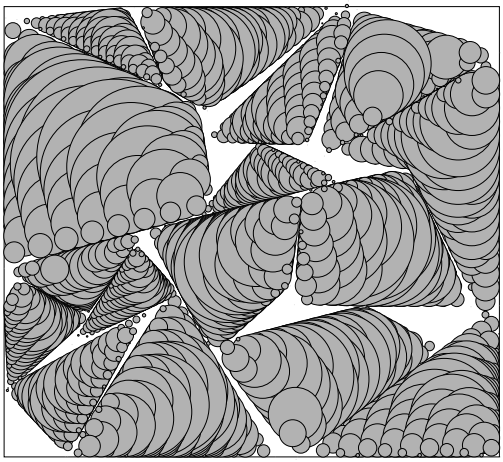
3.4.6 Experiments of Iterated Local Search

In this subsection, we compare the results of ILS_RIGID varying collision detection algorithms "AABB" and "SWEEP+SKIP." We set parameter $h = 3\bar{r}$ from the results in Section 3.4.5. We adopt a quasi-Newton method package L-BFGS with $m_{\mathrm{BFGS}} = 6$ in ILS_RIGID, and set the maximum number of iterations in each L-BFGS optimization to be 200. We conducted 10 runs under the time limit of 120 seconds per run. Since the sizes of the containers vary, we normalize the values of F_{rigid} by

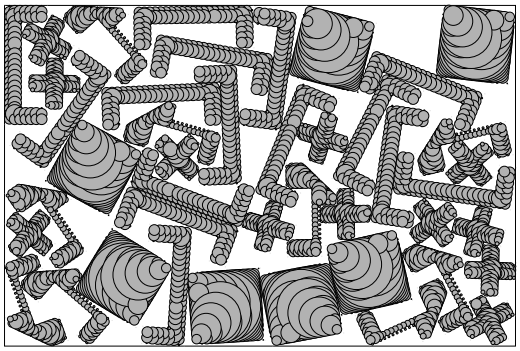
$$\frac{\sqrt{F_{\mathrm{rigid}}}}{\text{the average edge length of the container}}.$$

Table 3.5 shows the results of this experiment. The column "AQN" denotes the average number of the invocations of L-BFGS, the column "AOBJ" denotes the average value of F_{rigid} over the 10 runs, the column "BOBJ" denotes the best value of F_{rigid} over the 10 runs, the column "ANL" denotes the average normalized value of F_{rigid} over the 10 runs, and the column "BNL" denotes the best normalized value of F_{rigid} over the 10 runs.

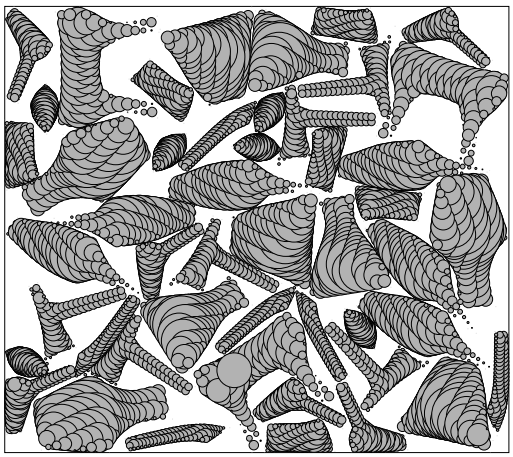
ILS_RIGID with the "SWEEP+SKIP" invoked L-BFGS more frequently than that with "AABB," and it obtained the best solutions for all instances. Figure 3.10 and 3.11 show the best solutions of all instances in 2D and 3D, respectively.



dighe1

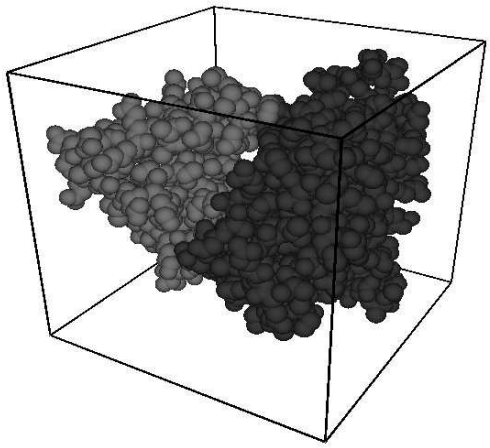


shapes0

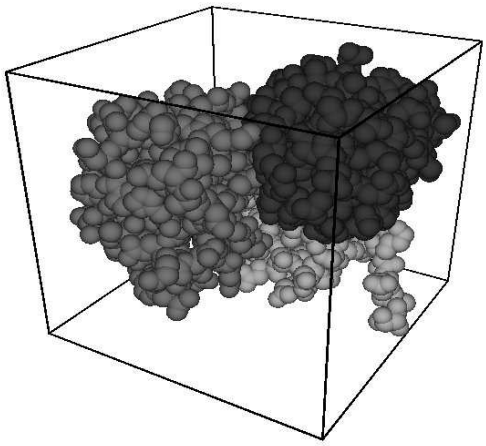


swim

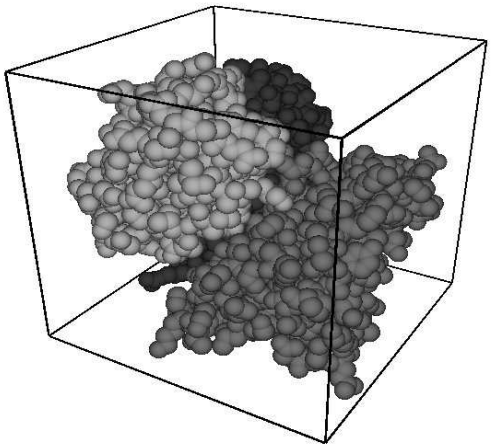
Figure 3.10: The best layouts for 2D instances.



mol1



mol2



mol3

Figure 3.11: The best layouts for 3D instances.

Table 3.5: The results of ILS_RIGID.

ILS_RIGID with AABB

Instance	AQN	AOBJ	BOBJ	ANL	BNL
dighe1	61	1.3e+02	4.1e+01	1.1e–01	6.1e–02
shapes0	29	2.7e+01	6.1e+00	1.0e–01	5.0e–02
swim	10	8.9e+05	1.6e+03	1.5e–01	6.4e–03
mol1	10	1.4e+01	1.3e+00	6.8e–02	2.1e–02
mol2	14	1.9e+01	1.2e+01	9.6e–02	7.4e–02
mol3	7	3.5e+01	4.4e+00	1.2e–01	4.1e–02

ILS_RIGID with SWEEP + SKIP

Instance	AQN	AOBJ	BOBJ	ANL	BNL
dighe1	205	2.7e+01	6.0e+00	5.0e–02	2.3e–02
shapes0	112	1.4e+00	2.8e–01	2.4e–02	1.0e–02
swim	60	1.5e+02	4.6e–11	2.0e–03	1.1e–09
mol1	95	1.2e+00	5.5e–01	2.0e–02	1.4e–02
mol2	127	7.1e+00	2.7e+00	5.8e–02	3.6e–02
mol3	93	5.3e+00	9.6e–01	4.5e–02	1.9e–02

3.4.7 Summary

We propose a new collision detection algorithm of sphere sets for the multi-sphere scheme. It divides a bounding box of all spheres into slabs and apply a plane sweep method to each slab. We also adopt a data structure to skip the self collisions. Our computational experiments indicate that our new algorithm runs considerably faster than the AABB based method and the iterated local search algorithm with COLDETECT2 can compute a layout with less penalty.

It is left as future work to compare our algorithm with other collision detection algorithms especially with algorithms with bounding volume hierarchies, and to incorporate the sphere bounding volume hierarchy in COLDETECT2 so as to extend our efficient implementation to the case with deformable objects.

3.5 Removing Overlaps in Label Layouts

3.5.1 Introduction

We apply the multi-sphere scheme to three problems arising from Graph Drawing. Our new approach is *flexible*, and has the following three advantages over previous work.

1. First, our approach can handle labels with *arbitrary shapes*. Note that previous methods can deal with only rectangular labels. However, in our approach, we can treat any non-rectangular-shaped labels by approximating each of them as a set of circles. We can also place given labels inside a specified area with a non-rectangular boundary.

2. Second, our algorithm can use three types of operation: *translation, translation with direction constraints* (i.e., move along the specified line), and *rotation*. Note that the previous methods deal with only translation.

3. Finally, our method can be used for *both 2D and 3D layouts*. Note that previous study can only deal with 2D layout.

In order to demonstrate our three advantages, we consider three new variations of the label overlap problem, each inspired by real world applications, and design an algorithm for each problem setting. More specifically, we use RigidQN for removing label overlaps with three different variations:

- **Problem 1:** rectangular labels with translation (inspired from metro map layout [50]),

- **Problem 2:** rectangular labels with direction constrained translation (inspired from road map layout [6]),

- **Problem 3:** 3D labels of arbitrary shape with both translation and rotation (inspired from 3D visualization with multiple attributes of nodes [7]).

We implemented our algorithm and evaluated with three different types of data sets. Our extensive experimental results show that our new approach is very fast and effective for removing label overlaps.

The remainder of the section is organized as follows. In Section 3.5.2, we formally define three new variations of label overlapping problem. Section 3.5.3 presents experimental results for two different real world applications. We conclude with future work in Section 3.5.4.

3.5.2 Problem Definition

The label overlap removal problem requires to remove a set of overlapping labels in the given layout by modifying the positions of the labels, so that no two labels overlap each other.

We now formally define three types of label overlap removal problem.

Problem 1: Rectangular Label with Translation

Input: A set of rectangles, where each rectangle is located on its initial position in the two-dimensional plane.

Output: A set of positions of rectangles such that no two rectangles overlap, and the change of new positions from the initial positions is small, where the new position of each rectangle is obtained by translation in any direction.

Problem 1 appears in applications such as placing labels of stations in a metro map layout [50].

Problem 2: Rectangular Label with Direction-Constrained Translation

Input: A set of overlapping rectangles, where each rectangle is located on its initial position with a specified direction constraints (i.e., a line segment) in the plane.

Output: A set of new positions of rectangles such that no two rectangles overlap and the change of new positions from the initial positions is small, where the new position of each rectangle is obtained by restricted translation along the specified direction only (i.e., on the line segment where the label initially lies).

Problem 2 appears in applications such as placing labels of street names in a road map layout [6].

Problem 3: 3D Multi-attribute Label with Translation and Rotation

Input: A set of overlapping spiked sphere (i.e., a sphere with several small cones on its surface), where each spiked sphere is located on its initial position in the 3D space.

Output: A set of new positions of spiked spheres in 3D such that no two spiked spheres overlap and the change of new positions from the initial positions is small, where the new position of each spiked sphere is obtained by both translation and rotation in 3D.

Problem 3 appears in applications such as visualization of network data with multiple attributes in three dimensions. For example, a spiked sphere was used to represent an author of the Information Visualization community, where each sphere represents an author, the size of sphere represents the number of research papers published by the author in the conference proceedings, and the length of each spike attached to the sphere represents special attributes such as the number of papers in specific research area [7].

We apply RigidQN to each type of label overlap removal problem by setting constraints of motions appropriately. Although we do not use an explicit criteria to minimize the total change between the initial and final layouts in RigidQN, it repeatedly modifies the initial layout by finding the best direction of translation of each object until a new layout with no overlap is obtained as a local optimal solution to our optimization problem. Thus in most cases, the final positions of labels are close to the initial positions.

3.5.3 Experimental Results

We conducted computational experiments of RigidQN by generating instances of Problem 1, 2 and 3. In this section, we report the results.

We implemented RigidQN in C++, compiled it by GCC 4.1 and conducted experiments on a PC with an AMD Sempron 3000+ 1.8 GHz processor and 450 MB memory. We adopted L-BFGS with $m_{\mathrm{BFGS}} = 6$.

Results of Problem 1

For the data set of Problem 1, we approximate each of the rectangles by finding a set of circles that cover the interior of the rectangle in order to ensure that the layout of the corresponding rectangles has no overlap whenever the penalty function has value 0.

There are many ways of choosing such a set of circles for each rectangle. Note that the use of small circles in approximation introduces less extra area outside the rectangle than the use of large circles. On the other hand, the use of large circles introduces a smaller number of variables in our optimization problem, thus leads to a faster running time than the use of small circles.

We tested with both options, a smooth approximation by small circles, and a rough approximation by large circles, to confirm the effect of the size of circles to the running time and quality of output.

Let r be a rectangle with height h and width w in a given layout, where we assume that h and w are multiples of 5.

In a smooth approximation, r is approximated with $(2w/2.5 - 1) \times (2h/2.5 - 1)$ circles

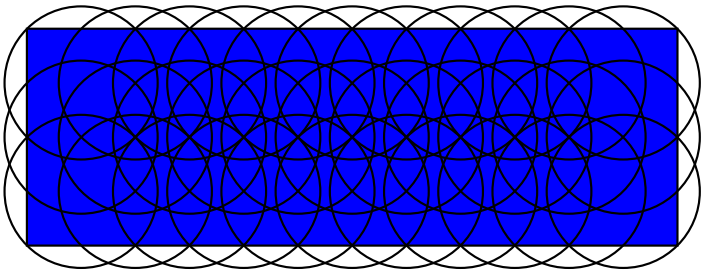


Figure 3.12: The smooth approximation of a rectangle by 33 circles for Problem 1.

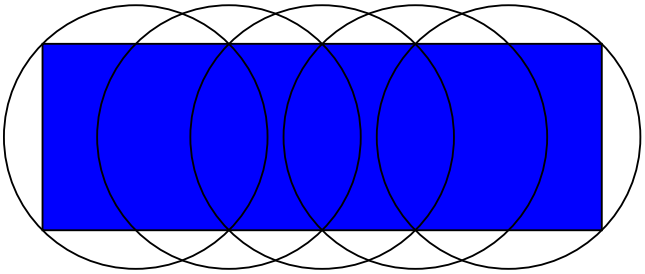


Figure 3.13: The rough approximation of a rectangle by 5 circles for Problem 1.

(see Figure 3.12). First, r is divided into $w/2.5 \times h/2.5$ squares of length 2.5. Each circle is circumscribed around each square of length 2.5, so that the label is contained by the generated circles. Next, new $(2w/2.5-1) \times (2h/2.5-1) - w/2.5 \times h/2.5$ circles are added to the circles generated in the first step. Each new circle is located between each adjacent generated circles.

In a rough approximation, r is approximated with $(2w/5-1) \times (2h/5-1)$ circles (see Figure 3.13). First, r is partitioned into $w/5 \times h/5$ squares of length 5. Each circle is circumscribed with each square of length 5, so that the label is contained by the generated circles. Next, new $(2w/5-1) \times (2h/5-1) - w/5 \times h/5$ circles are added to the circles generated in the first step. Each new circle is located between each adjacent generated circles.

Figures 3.12 and 3.13 show sets of circles generated for a rectangle of height 5 and of width 15 by the above two methods respectively. We can see that the smooth approximation gives a better approximation to the outer boundary of a rectangle.

For a real world data set of Problem 1, we used Tokyo metro map with 260 stations (i.e., labels). Based on the map, we generated an instance of 260 rectangles with the same width 15 and height 5. Figure 3.14 shows the initial layout of the instance. The layouts of the smooth approximation and the rough approximation have 8580 circles and 1300 circles, respectively.

Then, we applied RigidQN to the layouts of the smooth and rough approximations. Figures 3.15 and 3.16 show the final layouts of the Tokyo metro map with the smooth and

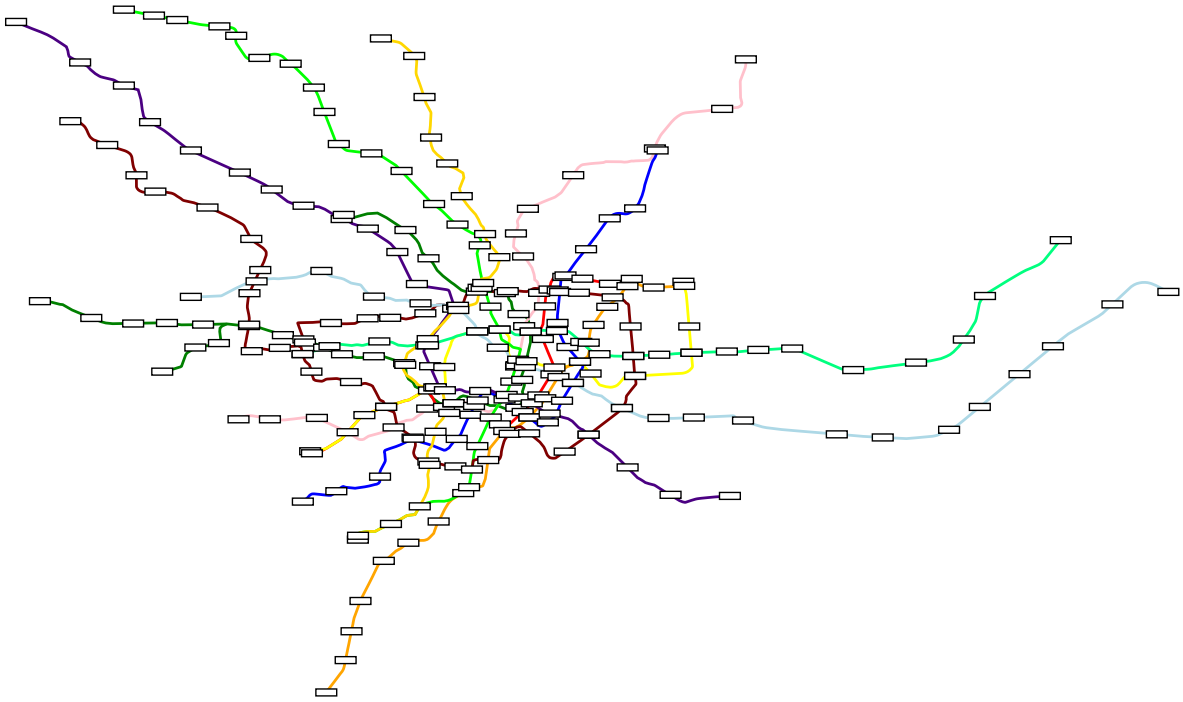


Figure 3.14: An initial layout of 260 rectangular labels of the metro map instance of Problem 1.

rough approximations, respectively. Note that RigidQN takes only 0.260 seconds for the smooth approximation and 0.056 seconds for the rough approximation.

Next, we used a road map with name labels of intersections in a part of Tokyo. As an initial layout of labels, we place a rectangular label at each intersection, where each label has the same height and a width which is proportional to the length of the corresponding intersection name (see Figure 3.17).

We approximated the labels with the rough approximation and applied RigidQN to the layout. Figure 3.18 shows the final layout of the rectangular labels. We observe that there is no overlap of the labels in the final layout. There are 410 labels and 4330 circles in the initial layout, and RigidQN takes 0.532 seconds to generate the final layout.

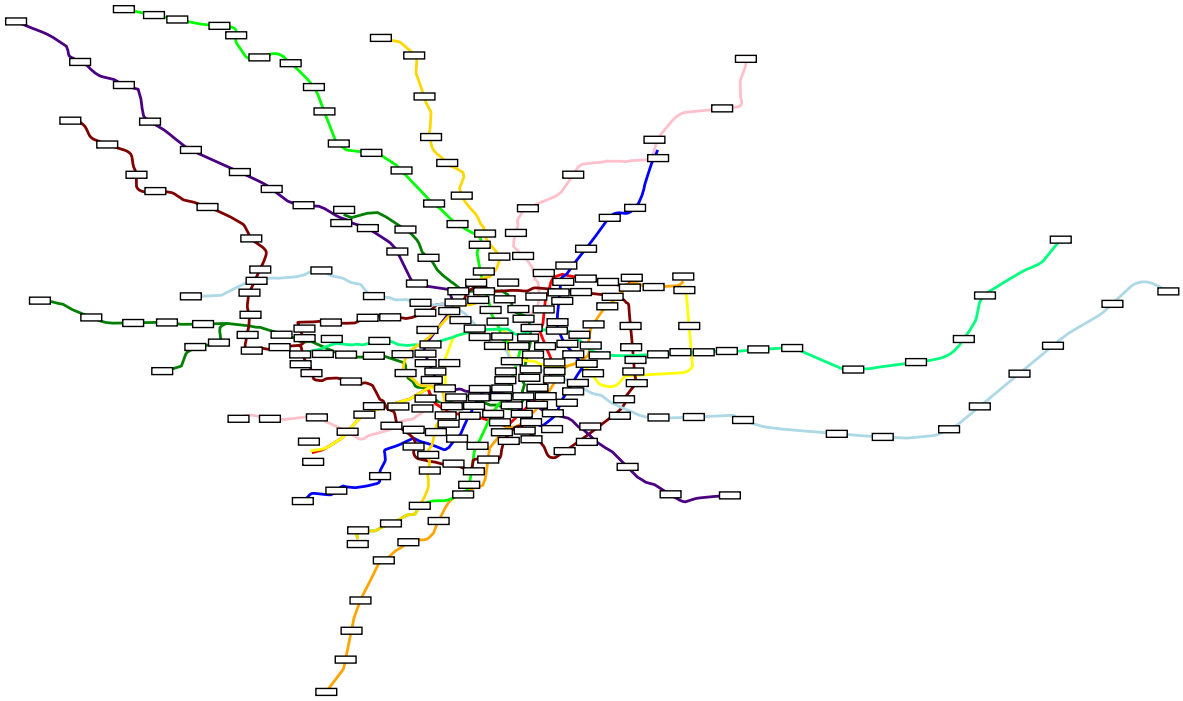


Figure 3.15: A final layout of the instance in Figure 3.14 with the smooth approximation.

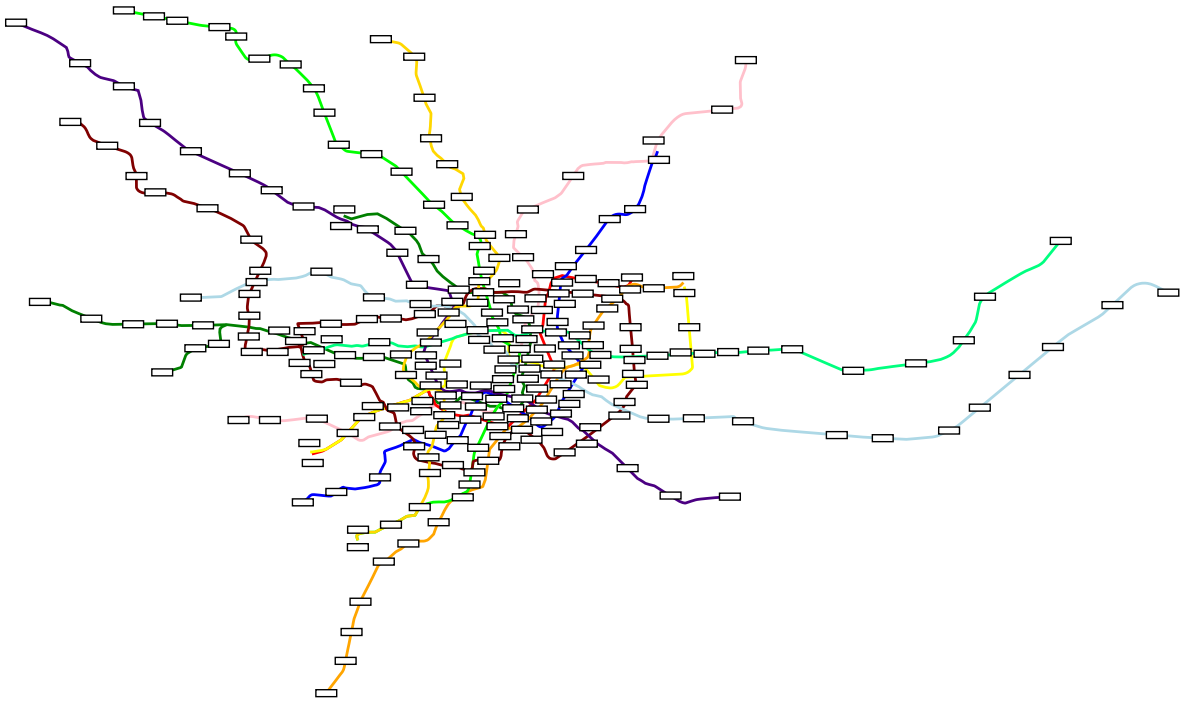


Figure 3.16: A final layout of the instance in Figure 3.14 with the rough approximation.

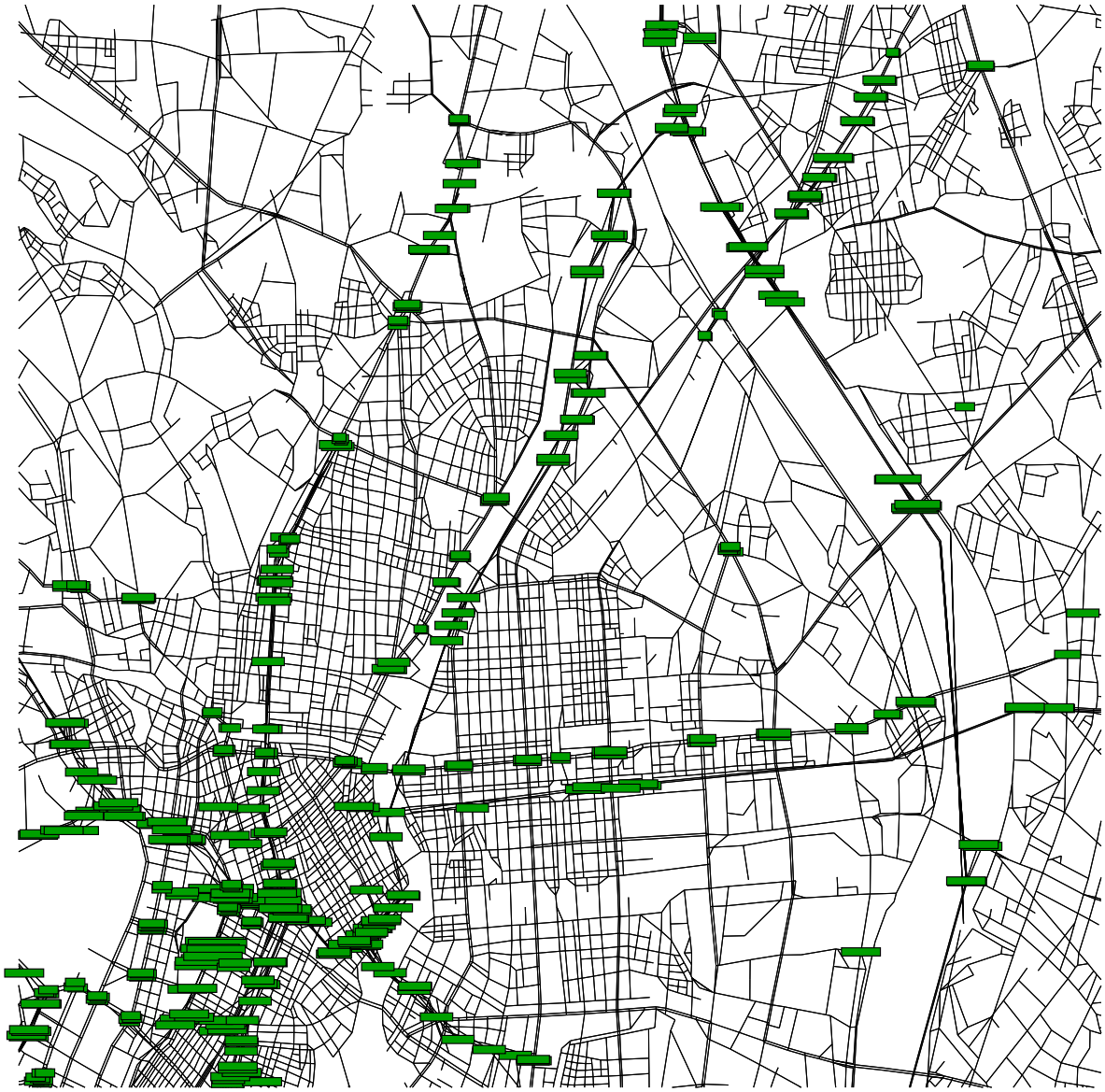


Figure 3.17: An initial layout of a road map with name labels of intersections.

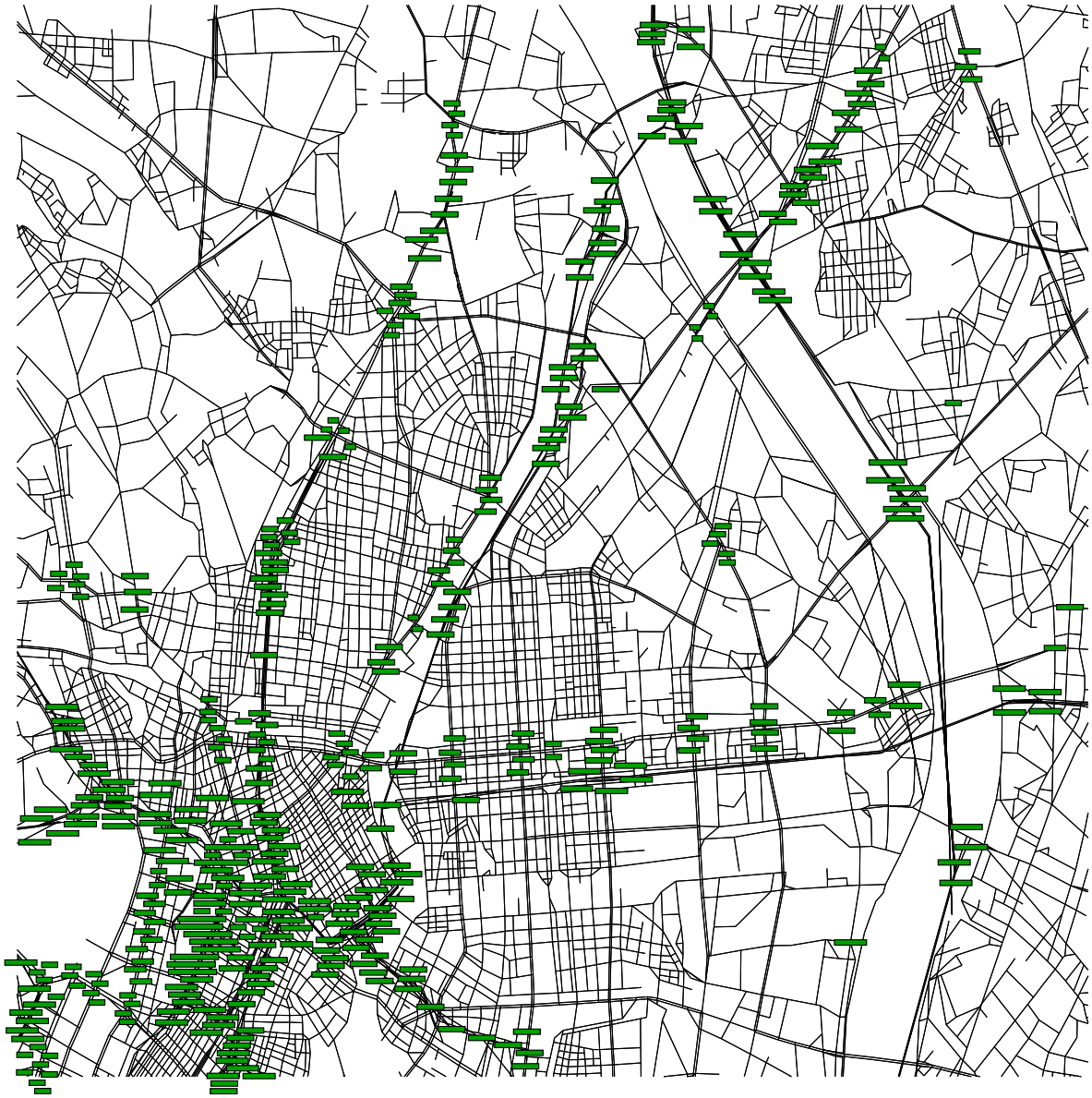


Figure 3.18: A final layout of the instance in Figure 3.17.

Results of Problem 2

In Problem 2, the movement of rectangular labels is limited to translation in specified directions, as appeared in the road maps. Thus, we generated instances which represent the road map label layout, where the labels (i.e., names of the streets) are constrained to be placed along the corresponding streets.

More specifically, we are given a set of edges embedded in the plane, where each edge is drawn as a straight-line segment that represents a street in the road map, and required to remove overlaps of rectangular labels by moving each label along each edge. Note that two edges (i.e., two streets) may have an intersection in the plane, in general.

The data set was generated as follows. We first start with a square with size $\ell_{\text{map}} \times \ell_{\text{map}}$ which consists of four lines as a drawing area, where we set $\ell_{\text{map}} = 10000$, and place a square grid on the square, where the minimum distance between two grid lines is ℓ_{grid}.

Next we draw horizontal and vertical line segments one after another on the grid lines. To draw a horizontal line segment, we choose two arbitrary vertical line segments whose distance is more than or equal to $\ell_{\text{map}}/3$ and connect them with an arbitrary horizontal line segment on the grid (we draw a vertical line segment analogously).

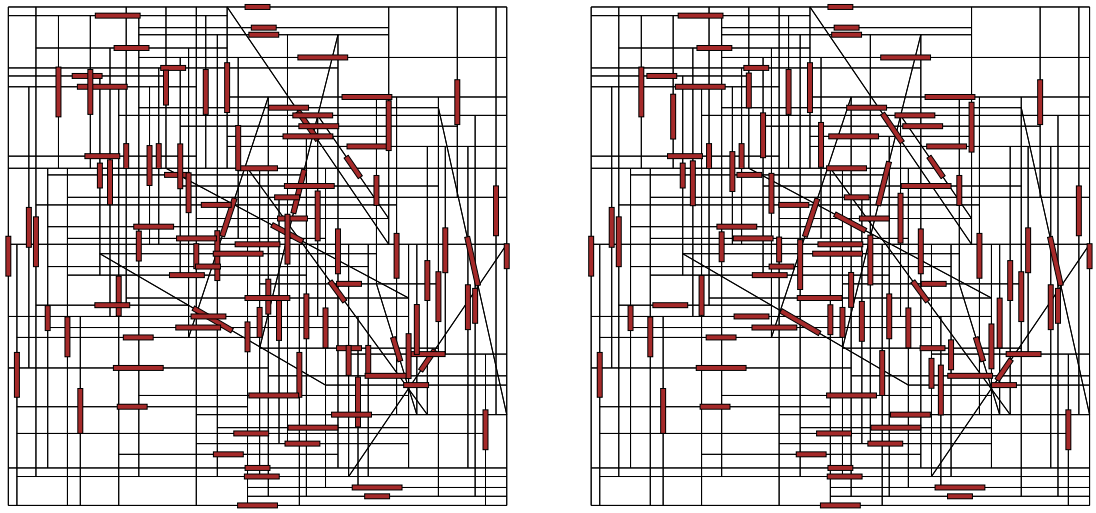
We repeat drawing line segments until we cannot choose any pair of line segments. Then, we draw some slanted line segments by choosing two arbitrary points in the drawing. Finally, we place a rectangle in the middle of each line segment in the drawing, where the height of a rectangle is ℓ_{label} and the length of a rectangle varies over a range $[5\ell_{\text{label}}, 10\ell_{\text{label}}]$.

For example, Figures 3.19(a), 3.20(a) and 3.21(a) show instances with initial positions of labels for $\ell_{\text{label}} = 100$, where a line segment represents an edge (i.e., street) and a rectangle represents a label (i.e., street name). Figure 3.19(a) is generated for $\ell_{\text{grid}} = 200$, which has 112 labels and 3601 circles. Figure 3.20(a) is generated for $\ell_{\text{grid}} = 150$, which has 147 labels and 4818 circles. Figure 3.21(a) is generated for $\ell_{\text{grid}} = 100$, which has 222 labels and 6773 circles.

Here we consider the problem of removing the overlap between all pairs of rectangles in the initial layout by moving each rectangle along the corresponding line segment. Again, we apply our algorithm RigidQN after approximating each rectangle by a set of circles.

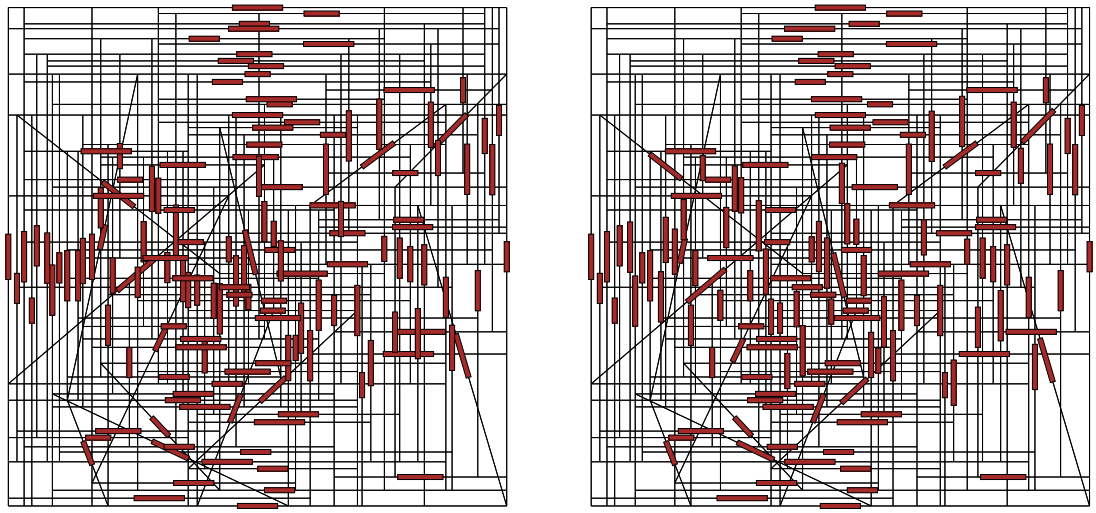
Note that for this approximation, we add more circles around the center of a rectangle. This is because we observed form our preliminary experiment that it was better than approximating a rectangle by placing circles equally. Thus, we place circles by increasing the number of circles from both ends of label as follows.

Assume that we are given a rectangle whose width w is longer than the height h and that



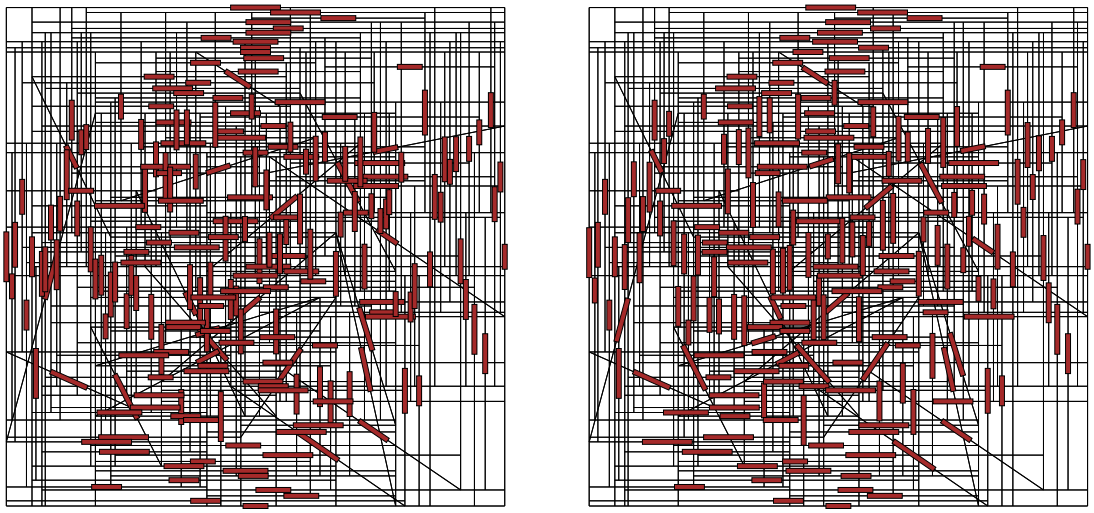
(a) (b)

Figure 3.19: An example of a road map layout with 112 labels ($\ell_{\text{label}} = 100, \ell_{\text{grid}} = 200$) : (a) an initial layout, (b) a final layout.



(a) (b)

Figure 3.20: An example of a road map layout with 147 labels ($\ell_{\text{label}} = 100, \ell_{\text{grid}} = 150$) : (a) an initial layout, (b) a final layout.



(a) (b)

Figure 3.21: An example of a road map layout with 222 labels ($\ell_{\text{label}} = 100, \ell_{\text{grid}} = 100$) : (a) an initial layout, (b) a final layout.

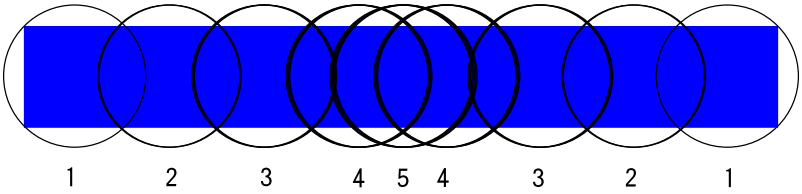


Figure 3.22: The approximation of a rectangle in road map instances.

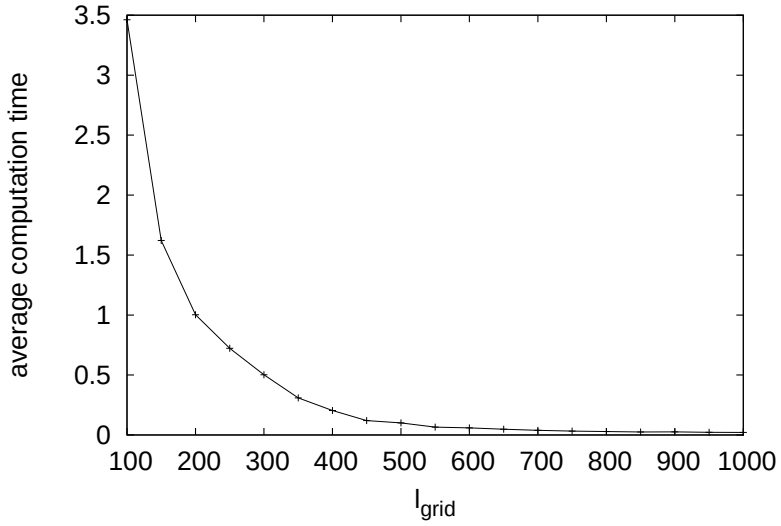


Figure 3.23: The average computation time for $\ell_{\text{label}} = 200$.

the bottom left point of the rectangle lies at the origin. We place circles with a radius $h/2$ at

$$\{((i-1/2)h, h/2), (w-(i-1/2)h, h/2) \mid i = 1, \ldots, \lfloor (w/h+1)/2 \rfloor\} \cup \{(w/2, h/2)\}.$$

The numbers of circles to place at $((i-1/2)h, h/2)$ and $(w-(i-1/2)h, h/2)$ are both i and the number of circles to place at $(w/2, h/2)$ is $\lceil (w/h+1)/2 \rceil$. See Figure 3.22 for an example. In the figure, the number under a circle represents the number of circles on the position.

See Figures 3.19(b), 3.20(b) and 3.21(b) for the final layouts of the instances of Figures 3.19(a), 3.20(a) and 3.21(a) after removing overlaps of labels by our method, respectively. It took 0.24 seconds for Figure 3.19(b), 0.43 seconds for Figure 3.20(b) and 1.5 seconds for Figure 3.21(b). Note that our method successfully removed all the overlaps for the sparse instance. However, for some dense instances, it left a few overlaps.

To observe the influence of the density of the road map layout and the number of labels on the efficiency of our algorithm, we varied two parameters ℓ_{label} and ℓ_{grid} from 100 to 1000 with a step size 50 and from 50 to 1000 with a step size 50, respectively, and conducted experiments. For each setting, we generate 100 instances and apply RigidQN to them.

Table 3.6 shows details of some of the the selected results. The column "time," "decrease," "#label," and "#circles" denote the average computation time, the average ratio of the decrease of the object function (3.1), the average number of labels, and the average number of circles, respectively. Figures 3.23 and 3.24 show the average computation time and the average ratio of the penalty decrease for $\ell_{\text{label}} = 200$, respectively. We observed that our algorithm removed almost all overlaps in less than one second for the instances for $\ell_{\text{grid}} \geq 2\ell_{\text{label}}$.

Table 3.6: Details of the selected results of road map instances.

ℓ_{label}	ℓ_{grid}	time	decrease (%)	#labels	#circles
100	250	0.12	99.9	9.0×10^1	2.8×10^3
	200	0.22	99.6	1.1×10^2	3.4×10^3
	150	0.65	98.8	1.4×10^2	4.5×10^3
	100	1.57	98.3	2.2×10^2	6.9×10^3
200	400	0.20	98.6	5.7×10^1	1.7×10^3
	350	0.30	98.3	6.3×10^1	1.9×10^3
	300	0.50	97.0	7.4×10^1	2.3×10^3
	250	0.72	95.5	9.0×10^1	2.8×10^3
300	600	0.16	97.4	3.7×10^1	1.1×10^3
	500	0.28	96.4	4.6×10^1	1.4×10^3
	400	0.48	92.4	5.7×10^1	1.7×10^3
	300	0.79	83.5	7.4×10^1	2.3×10^3
400	1000	0.07	98.0	2.4×10^1	7.5×10^2
	800	0.12	94.6	2.8×10^1	8.8×10^2
	600	0.29	89.9	3.7×10^1	1.1×10^3
	400	0.66	74.6	5.7×10^1	1.7×10^3
500	1000	0.12	89.4	2.4×10^1	7.5×10^2
	800	0.19	87.4	2.8×10^1	8.7×10^2
	600	0.32	81.1	3.7×10^1	1.1×10^3
	400	0.71	65.5	5.7×10^1	1.8×10^3

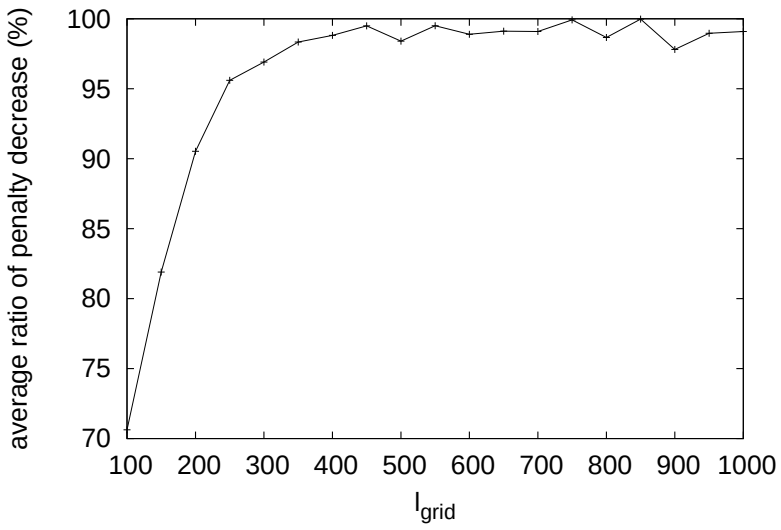
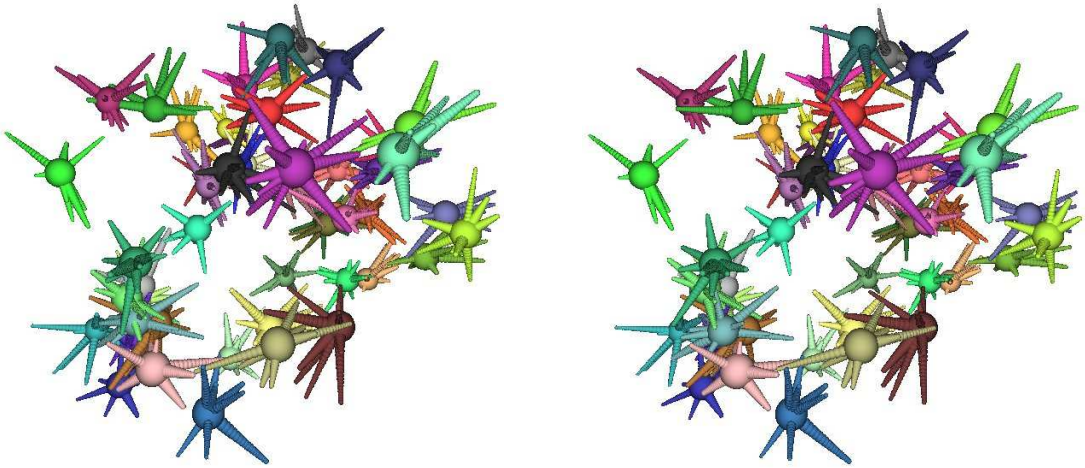


Figure 3.24: The average ratio of penalty decrease for $\ell_{\text{label}} = 200$.



(a) (b)

Figure 3.25: An example instance with 50 labels of Problem 3: (a) an initial layout, (b) a final layout.

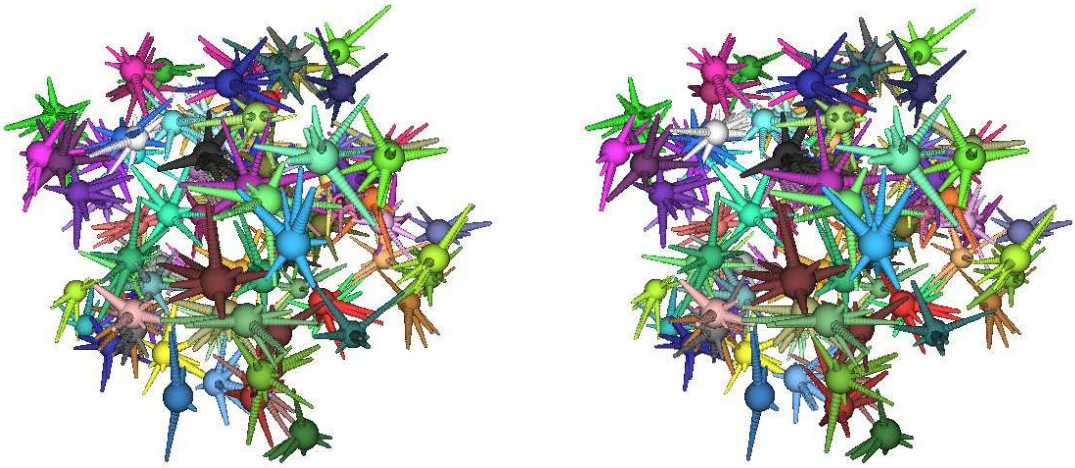
Results of Problem 3

For the data set of Problem 3 (i.e., 3D objects of various shapes with rotation and translation), we create an instance which resembles the spiked spheres used in [7]. More specifically, given a layout of a set of spiked spheres with some overlaps, we remove the overlaps by translation and rotation.

We generate an instance as follows. A spiked sphere has a sphere of radius 10 together with attached 10 spikes. Each spike consists of 20 spheres and the length varies on a range $[10, 70]$. Thus a spiked sphere has 201 spheres in total. To create an instance, we place the spiked spheres randomly in a cube with edge length 300, where the number of spiked spheres is a parameter.

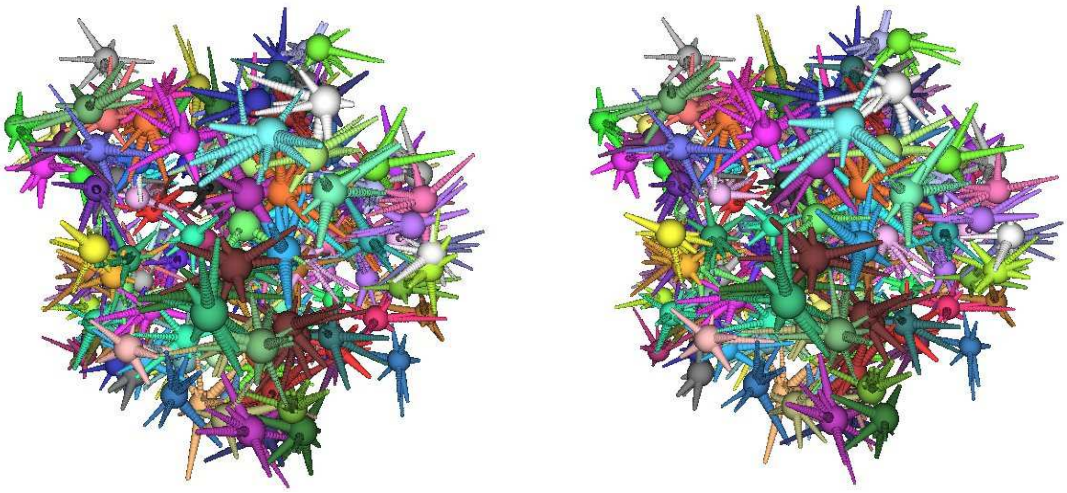
See Figures 3.25(a), 3.26(a) and 3.27(a) for instances with 50, 100 and 200 spiked spheres, respectively. See Figures 3.25(b), 3.26(b) and 3.27(b) for the resulting layouts. Our algorithm RigidQN ran in 0.47 seconds for Figure 3.25(b), 1.7 seconds for Figure 3.26(b) and 8.8 seconds for Figure 3.27(b), and obtained layouts with no overlap of spiked spheres by translating and rotating them slightly. Figures 3.28(a) and (b) are magnified pictures of Figures 3.26(a) and (b), respectively. We can see a spiked sphere in Figure 3.28(a) penetrating another spiked sphere, and the removal of overlap in Figure 3.28(b).

To observe the influence of the number of spiked spheres on the efficiency of our algorithm, we varied the number of spiked spheres from 50 to 250 with a step size 50, generated 10 instances for each setting, and measured the computation time. See Figure 3.29 for the average computation time. In this experiment, RigidQN found a layout with an objective



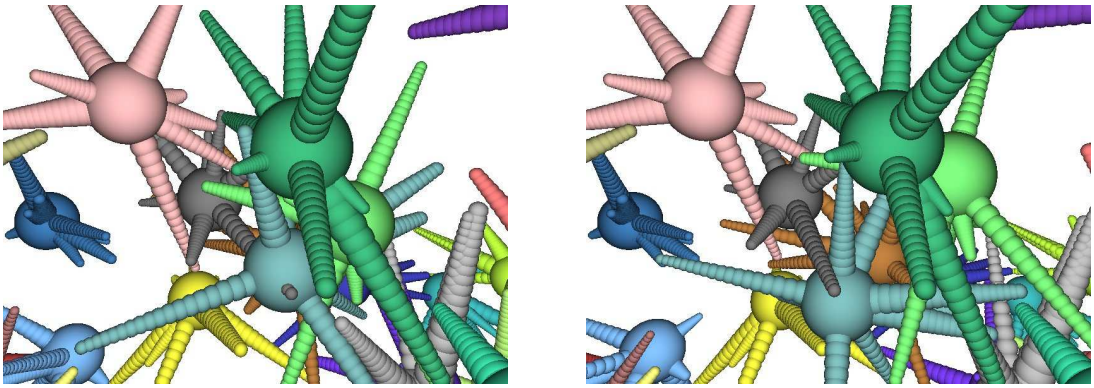
(a) (b)

Figure 3.26: An example instance with 100 labels of Problem 3: (a) an initial layout, (b) a final layout.



(a) (b)

Figure 3.27: An example instance with 200 labels of Problem 3: (a) an initial layout, (b) a final layout.



(a) (b)

Figure 3.28: Magnified figures of Figure 3.26: (a) an initial layout, (b) a final layout.

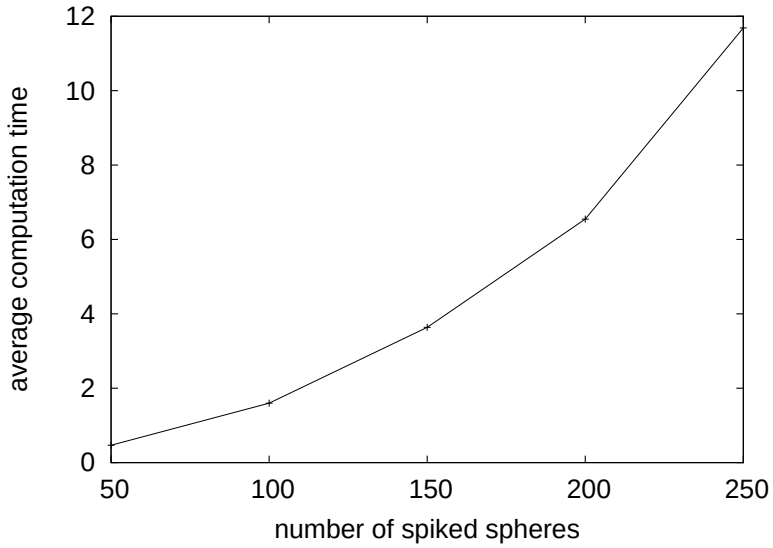


Figure 3.29: The average computation time for instances of Problem 3.

function value less than 10^{-9} for all instances. We observed that our algorithm removed almost all overlaps in less than 10 seconds for the instances with the number of spikes spheres less than or equal to 200.

3.5.4 Summary

We presented a new approach for two new variations of label overlap removal problem based on multi-sphere scheme. Our approach is flexible to support various operations such as translation, translation with direction constraints, and rotation. Further, our method can support labels with arbitrary shapes in both 2D and 3D layout settings.

We applied our algorithm to two new label overlap problems: two dimensional rectan-

gular label with translation (Problem 1), two dimensional rectangular label with directed-constrained translation (Problem 2), and three dimensional multi-attribute label with translation and rotation (Problem 3). The experimental results showed that our algorithm based on local search algorithm RigidQN was very efficient for label overlap removal.

For Problem 1, RigidQN removed all overlaps of the labels with several thousand circles in less than one second. For Problem 2, RigidQN removed almost all overlaps of the labels if the density of labels in the initial layout was not so high, and found a layout in less than one second for the instances with a few thousand circles. For Problem 3, it also removed almost all overlaps of the labels in less than 10 seconds for the instances with less than or equal to 20000 spheres.

Our future work includes more extensive experiments with real world applications. For example, we will consider label overlap removal problem with domain-specific layout constraints including software engineering (such as UML diagram), biology (such as biochemical pathways), and social networks visualization.

Chapter 4

Conclusion

Throughout this thesis, we have considered the developments of solvers for cutting and packing problems. The studies in this thesis are summarized as follows.

First, in Chapter 2 we considered the two-dimensional irregular strip packing problem, which allows each polygon to rotate by fixed degrees. We develop a two-layered algorithm: the outer layer finds the minimum length of the strip and the inner layer finds a feasible layout of polygons into the container. We proposed an iterated local search algorithm based on nonlinear programming for the inner layer. We allow overlaps of polygons and protrusions of polygons from the container in a layout being modified and penalize them to formulate the overlap minimization problem. By defining the penalty using the penetration depth, we also introduced the polygon separation problem as an unconstrained nonlinear programming problem that has a differentiable objective function. We realized a separation algorithm by applying L-BFGS method to the polygon separation problem. We also develop an algorithm to translate a polygon into the position such that the total penalty is minimized and realized a swapping operation of polygons based on the translation algorithm as the perturbation of locally optimal solutions in our iterated local search algorithm. Through computational experiments using benchmark instances, we showed that our method is competitive with the existing algorithms and updated the best known solutions for several benchmark instances.

Then, in Chapter 3 we proposed the general framework *multi-sphere scheme* for cutting and packing problems. In the multi-sphere scheme, we consider the problem that asks to find a feasible layout of objects in two or three dimensional space. We first approximate objects by sphere set and then find a layout of objects. In general, it is computationally hard to check whether two objects in arbitrary shapes overlap or not and compute how much they overlap each other. However, approximated objects by sphere sets make the implementation of the operations computationally easy and robust. Besides, we can handle free rotations of objects by approximating objects by spheres. In this thesis, we concentrate on finding a feasible lay-

out. We formulated the penalty minimization problem, which allows penetrations of spheres that belong to different objects and protrusions of spheres from the container and penalizes them. We showed that we can handle some constraints on motions of objects, e.g., "translations," "translations in fixed directions," and "translations and rotations." We designed the penalty minimization problem as an unconstrained nonlinear programming problem such that the objective function is differentiable as in Chapter 2. We realized a local search algorithm by applying L-BFGS method to the penalty minimization problem. Also we proposed an iterated local search algorithm incorporated the local search algorithm and a swapping operation as a perturbation.

From our preliminary experiments, we observed that the collision detection of spheres was often the bottle neck of the entire algorithm. We proposed a fast collision detection algorithm of spheres that divides a bounding box of spheres into slabs and applies the plane sweep method to each slab. In Section 3.3, we showed that our collision detection algorithm runs in $O(n \log n + K)$ time with $O(n + K)$ space under the assumption that the dimension and the maximum ratio of radii are constant, where n is the number of spheres and K is the number of colliding pairs of spheres. Then, in Section 3.4 we proposed a simpler collision detection algorithm for the implementation and conducted the computational results compared with those by the axis-aligned bounding box based algorithm. We observed that the new algorithm runs much faster than the other.

Finally in Section 3.5 we applied the multi-sphere scheme to removal of overlaps in label layouts. This problem arises from graph drawing. We consider three variations: metro map (each label translates in two dimension), road map (each label translates along a prescribed direction in two dimension), and spiked sphere (each object translates and rotates in three dimension). Computational results showed that our method can find feasible layouts for all problems quickly. This means that our method is flexible and effective for removing overlaps in label layouts.

Our future work on completion of the multi-sphere scheme is described as follows.

- Approximation of objects by sets of spheres.
  In this thesis, we designed the approximation algorithms depending on the problems. In order to apply the multi-sphere scheme to various problems, we need to provide an algorithm that approximates objects in a general shape. It is preferable to approximate objects with a set of spheres such that the number of spheres is small and the maximum ratio of two spheres is small. Such a choice of spheres would reduce the running time according to the analysis of the time complexity of the collision detection in Section 3.3. We can approximate an object by placing small spheres around the boundary of the

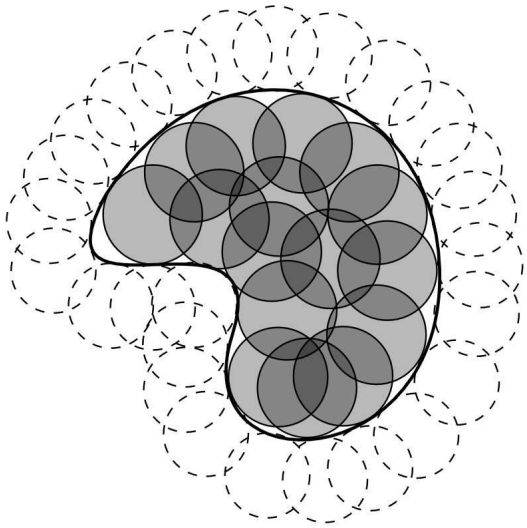


Figure 4.1: Approximating an object by filling spheres.

object and then filling the interior of the boundary with spheres by the multi-sphere scheme itself, where we penalize only the penetrations between the fixed spheres on the boundary and the spheres to be filled. See Figure 4.1 for an example. Dashed circles represent the fixing circles and grey circles represent the filled circles.

- Other primitive tools.
  In this thesis, we used spheres to represent objects in the multi-sphere scheme. To make the scheme more flexible, we have to adopt other types of shapes rather than usual spheres as primitive shapes to approximate given objects. Among possible other shapes are ellipsoids and negative spheres, where a negative sphere is the complement of a usual sphere.

  In the penalized rigid sphere set packing problem (3.1), we added penetration penalties of objects and protrusion penalties of objects from a container. In our nonlinear programming formulation, it is easy to specify whether a penetration penalty is imposed or not for each pair of spheres, and to incorporate other kinds of penalties into the objective function. The extension is useful in handling more flexible problem settings such as deformable objects and objects with movable regions. As such an example of penalties, we propose *distance range penalties* as follows. A distance range penalty has a range $[l_{ij}, u_{ij}]$ and a pair (S_i, S_j) of sphere as parameters. Let $\boldsymbol{c}_i$ and $\boldsymbol{c}_j$ be the centers of S_i and S_j, respectively. Then, the distance range penalty $f^{\mathrm{dis}}(S_i, S_j)$ of S_i and S_j in

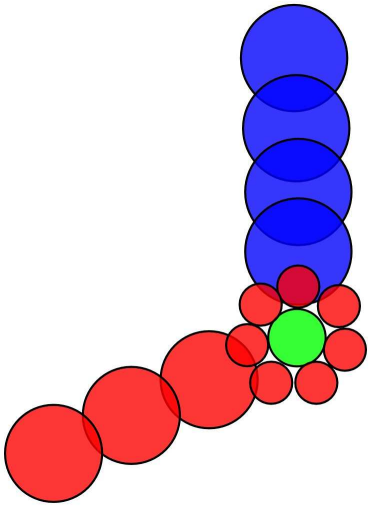


Figure 4.2: A joint of two objects.

a layout is defined by

$$f^{\mathrm{dis}}(S_i, S_j) = (\min\{l_{ij} - \|\boldsymbol{c}_i - \boldsymbol{c}_j\|, 0\} + \max\{\|\boldsymbol{c}_i - \boldsymbol{c}_j\| - u_{ij}, 0\})^2.$$

We can specify the region within which an object can translate or rotate by using the sine function and the cosine function in the motion functions of objects, because the range of a function $f(\theta) = a \sin\theta + b$ is limited by $[b - a, b + a]$. Other motions such as translation around the boundary of a circle/sphere may be useful for various layout problems in Graph Drawing or Information Visualization.

- Flexible objects.
  In this thesis, we focused on the problem where objects are rigid. However, there are many real world applications that involve flexible objects such as objects with joints. Adding a deformation penalty to the penetration penalty is a possible way of dealing with such problems. However, this would cause difficulty in maintaining layouts without penetration due to the mixture of two different penalties. Hence we need to seek for alternative methods depending on the features of problems.

  Figure 4.2 illustrates an example of a joint of two objects. An object consists of the red circles and the other consists of the blue and green circles. If we penalize the penetrations only between the green circle and the red circles (no penetration penalty between the red circles and the blue circles), then the two objects will behave like a joint. We can also realize a joint by adding distance range penalties between two objects (see Figure 4.3). The two arrows connect the red object and the blue object. It is also

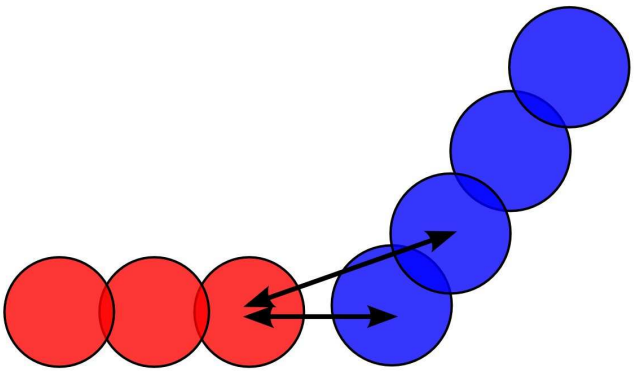


Figure 4.3: Another type of a joint of two objects.

possible to control the range of the angle of the joint by defining the distance range penalties in an appropriate way.

When we try to remove the overlap of two intersecting objects by quasi-Newton method, the resulting layout often leaves an unnecessarily large space between them. We can prevent overlapping objects from moving far away in the local search procedure by using a soft *cushion*. For example, see the blue and red objects in Figure 4.4. Suppose that we want to detect a layout with no overlapping and without a large space between the two objects. If we apply the local search to the layout directly, they may get away. To prevent this, we add a cushion around the blue objects. The cushion consists of grey circles, where each grey circle moves independently and we penalize the penetrations between the blue object and the grey circles (no penetration penalty among the grey circles). The cushion will play as resistance to the motion of the blue object against the red object.

- Cost functions.
  The objective function of the formulation (3.1) of the multi-sphere scheme contains only the penalties of penetrations and protrusions. However, there are some applications which can be formulated as a problem of finding the best layout of given objects among all overlap-free layouts, where a cost function is usually used to measure the quality of a layout. For this, we need to extend the multi-sphere scheme to deal with cost functions. It seems useful to utilize constrained optimization methods rather than L-BFGS to achieve the extension.

- Motion planning.
  Motion planning is a problem to find a tour of an object (called *robot*) from a start

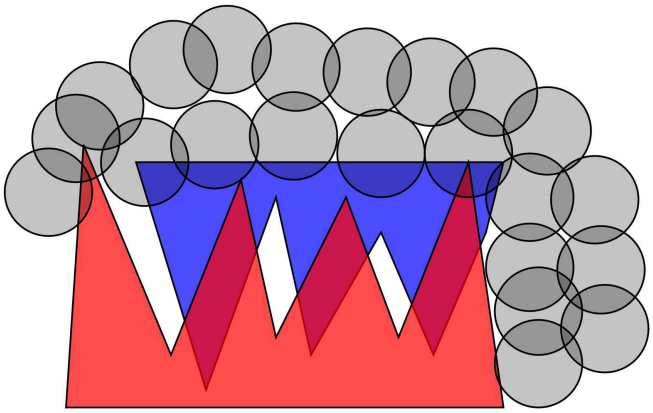


Figure 4.4: A cushion to prevent the blue object moving against the red object.

to a goal such that the object do not collide with obstacles. One of the representative algorithms for the motion planning is the *probabilistic roadmap method* [22, 61], which randomly places the robot in the space many times, generates a graph by connecting the positions at which the robot collides with no obstacle, and finds a tour from the graph. In the phase to place the robot randomly, we can remove the overlap between the robot and the obstacles by applying the local search of the multi-sphere scheme. This may generate more feasible positions of the robot and make the graph connecting the positions more informative.

We also search for more real world applications to which we apply the multi-sphere scheme to know necessary improvements on the current multi-sphere scheme.

Appendix A

Unconstrained Nonlinear Programming

Unconstrained nonlinear programming problem is an unconstrained optimization problem with a nonlinear objective function. Since nonlinear programming plays an important role in our algorithms in this thesis, we review it here. See textbooks by Nocedal and Wtight [80], and Fletcher [36] for more details. An n-dimensional unconstrained nonlinear programming problem is formally described as

$$\begin{aligned} \text{minimize} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & \boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n, \end{aligned}$$

where the objective function is $f : \mathbb{R}^n \to \mathbb{R}$.

If function f is continuously differentiable at $\boldsymbol{x} \in \mathbb{R}^n$, the *gradient* of f at $\boldsymbol{x}$ is defined by

$$\nabla f(\boldsymbol{x}) = \left(\frac{\partial f}{\partial x_1}(\boldsymbol{x}), \ldots, \frac{\partial f}{\partial x_n}(\boldsymbol{x}) \right)^\mathsf{T}.$$

If function f is twice continuously differentiable at $\boldsymbol{x} \in \mathbb{R}^n$, the *Hessian* of f at $\boldsymbol{x}$ is defined by

$$\nabla^2 f(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1}(\boldsymbol{x}) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(\boldsymbol{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(\boldsymbol{x}) & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n}(\boldsymbol{x}) \end{pmatrix}.$$

In the case where the gradient of the objective function is available, we can find a locally optimal solution effectively by applying *quasi-Newton method*, *conjugate gradient method* and *limited memory BFGS method*. If both the gradient and the Hessian of the objective function are available, then we can apply *Newton's method*. Their scheme is similar. We iteratively compute a direction, a step size of the direction, and update a solution by going forward into the direction by the step size. The scheme is formally described in Algorithm 11, Note that

this scheme is based on a *line search method*. *Trust region method* is also incorporated in the above methods, but we focus only on the line search method in this section.

Algorithm 11 : Basic scheme for unconstrained optimization problems

Given an initial solution $\boldsymbol{x}_0$ and a terminate condition ε;
$k := 0$;
while $\|\nabla f(\boldsymbol{x}_k)\| > \varepsilon$ **do**
  Compute a direction $\boldsymbol{d}_k$;
  Find a step size α_k such that $f(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k) < f(\boldsymbol{x}_k)$;
  $k := k + 1$
end while.

The main difference of the nonlinear programming methods is how to compute the search direction $\boldsymbol{d}_k$. In the *steepest descent method*, which is a basic method, we just let $\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k)$. But the others compute the direction in more sophisticated ways. We shortly review the methods for search directions in Section A.2

A.1 Line Search Methods

Line search is a procedure to try to find a step size α_k such that

$$\min_{\alpha \geq 0} f(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k).$$

It is iteratively applied in Algorithm 11 to unconstrained nonlinear optimization problems. It is categorized as *exact line search* or *inexact line search*. Exact line search compute the optimal step size. On the other hand, inexact line search usually choose a step size α_k such that the descent of the objective function $f(\boldsymbol{x}_k) - f(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k) < 0$ is enough large for the user. The exact line search has advantage in the theoretical point of view. However, it tends to cost more computation time than inexact line search does. Therefore, inexact line search is popular in practice. For an example of inexact line search methods, we review Armijo rule in Algorithm 12.

Algorithm 12 : Armijo rule

Given $\beta \in (0, 1)$, $\rho \in (0, 1/2)$, $c > 0$;
$\alpha_k := c\beta$;
while $f(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k) \leq f(\boldsymbol{x}_k) + \rho \alpha_k \nabla f(\boldsymbol{x}_k)^\mathsf{T} \boldsymbol{d}_k$ **do**
  $\alpha_k := \alpha_k \beta$
end while;
Return α_k.

A.2 Search Directions

In this section, we let $\boldsymbol{g}_k = \nabla f(\boldsymbol{x}_k)$ and $G_k = \nabla^2 f(\boldsymbol{x}_k)$ for simplicity.

Newton's Method

In Newton's method, we let the search direction at $\boldsymbol{x}_k$ by

$$\boldsymbol{d}_k = -G_k^{-1}\boldsymbol{g}_k.$$

It is known to converge quadratically if some conditions are satisfied. However it costs $O(n^3)$ time to obtain $\boldsymbol{d}_k$ and not suitable for large scale problems.

Quasi-Newton Method

Let $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$ and $\boldsymbol{y}_k = \boldsymbol{g}_{k+1} - \boldsymbol{g}_k$. DFP method and BFGS method are well-known for quasi-Newton method and BFGS method is popular in practice. We review BFGS method. It has an approximation matrix H_k of the Hessian G_k. Initially, we let H_1 be an identity matrix. We compute the search direction at $\boldsymbol{x}_k$ by

$$\boldsymbol{d}_k = -H_k\boldsymbol{g}_k,$$

and update H_k by

$$H_{k+1} = H_k - \left(\frac{\boldsymbol{s}_k\boldsymbol{y}_k^\mathsf{T}H_k + H_k\boldsymbol{y}_k\boldsymbol{s}_k^\mathsf{T}}{\boldsymbol{s}_k^\mathsf{T}\boldsymbol{y}_k}\right) + \frac{\boldsymbol{s}_k\boldsymbol{s}_k^\mathsf{T}}{\boldsymbol{s}_k^\mathsf{T}\boldsymbol{y}_k}\left(1 + \frac{\boldsymbol{y}_k^\mathsf{T}H_k\boldsymbol{y}_k}{\boldsymbol{s}_k^\mathsf{T}\boldsymbol{y}_k}\right).$$

It is known to converge superlinearly. However, since it requires $O(n^2)$ space to keep H_k and costs $O(n^2)$ time to obtain $\boldsymbol{d}_k$, it is not suitable for large scale problems.

Conjugate Gradient Method

Various methods are proposed for the conjugate gradient method, e.g.,

$$\boldsymbol{d}_k = -\boldsymbol{g}_k + \beta_{k-1}\boldsymbol{d}_{k-1},$$

where

$$\beta_{k-1} = \frac{\boldsymbol{g}_k^\mathsf{T}\boldsymbol{g}_k}{\boldsymbol{g}_{k-1}^\mathsf{T}\boldsymbol{g}_{k-1}}, \qquad \text{(Fletcher and Reeves formula)}$$

$$\beta_{k-1} = \frac{\boldsymbol{g}_k^\mathsf{T}(\boldsymbol{g}_k - \boldsymbol{g}_{k-1})}{\boldsymbol{g}_{k-1}^\mathsf{T}\boldsymbol{g}_{k-1}}. \qquad \text{(Polak and Ribiere formula)}$$

Since it takes only $O(n)$ time to compute $\boldsymbol{d}_k$, it can be applied to large scale problems. However, it does not converge quickly compared with Newton's method and quasi-Newton method.

Limited Memory BFGS Method

Liu and Nocedal [72] proposed *Limited Memory BFGS (L-BFGS) method* for large scale problems. We keep recent m pairs $(\boldsymbol{s}_i, \boldsymbol{y}_i)$ $(i = k-m, \ldots, k-1)$ to compute H_k.

$$\begin{aligned} H_k = {} & (V_{k-1}^{\mathsf{T}} \ldots V_{k-m}^{\mathsf{T}}) \frac{\boldsymbol{s}_k^{\mathsf{T}} \boldsymbol{y}_k}{\|\boldsymbol{y}_k\|^2} I (V_{k-m} \ldots V_{k-1}) \\ & + \rho_{k-m} (V_{k-1}^{\mathsf{T}} \ldots V_{k-m+1}^{\mathsf{T}}) \boldsymbol{s}_{k-m} \boldsymbol{s}_{k-m}^{\mathsf{T}} (V_{k-m+1} \ldots V_{k-1}) \\ & + \rho_{k-m+1} (V_{k-1}^{\mathsf{T}} \ldots V_{k-m+2}^{\mathsf{T}}) \boldsymbol{s}_{k-m+1} \boldsymbol{s}_{k-m+1}^{\mathsf{T}} (V_{k-m+2} \ldots V_{k-1}) \\ & + \ldots \\ & + \rho_{k-1} \boldsymbol{s}_{k-1} \boldsymbol{s}_{k-1}^{\mathsf{T}}, \end{aligned}$$

where

$$\rho_k = \frac{1}{\boldsymbol{s}_k \boldsymbol{y}_k^{\mathsf{T}}}, \quad V_k = I - \rho \boldsymbol{y}_k \boldsymbol{s}_k^{\mathsf{T}}.$$

L-BFGS method also converges slowly compared with Newton's method and quasi-Newton method. However, it requires only $O(n)$ space and $O(n)$ time under the assumption that m is constant and suitable for large scale problems. In fact $3 \leq m \leq 7$ is recommended by Liu and Nocedal [72]. We adopt L-BFGS method in our algorithms in this thesis, where we let $m = 6$ in our experiments.

Bibliography

[1] Aarts, E., and Lenstra, J. K. *Local Search in Combinatorial Optimization*. Princeton University Press, 2003.

[2] Adamowicz, M., and Albano, A. Nesting two-dimensional shapes in rectangular modules. *Computer-Aided Design 8*, 1 (1976), 27–33.

[3] Agarwal, P., Guibas, L., Nguyen, A., Russel, D., and Zhang, L. Collision detection for deforming necklaces. *Computational Geometry 28*, 2-3 (2004), 137–163.

[4] Agarwal, P. K., Guibas, L. J., Har-Peled, S., Rabinovitch, A., and Sharir, M. Penetration depth of two convex polytopes in 3D. *Nordic Journal of Computing 7*, 3 (2000), 227–240.

[5] Aggarwal, A., Guibas, L. J., Saxe, J., and Shor, P. W. A linear time algorithm for computing the voronoi diagram of a convex polygon. In *STOC 1987: Proceedings of the 19th Annual ACM Conference on Theory of Computing* (1987), ACM Press, pp. 39–45.

[6] Agrawala, M. *Visualizing Route Maps*. PhD thesis, Stanford University, 2002.

[7] Ahmed, A., Dwyer, T., Hong, S.-H., Murray, C., Song, L., and Wu, Y. X. Visualisation and analysis of large and complex scale-free networks. In *EUROVIS 2005: Eurographics / IEEE VGTC Symposium on Visualization* (2005), K. Brodlie, D. Duke, and K. Joy, Eds., Eurographics Association, pp. 239–246.

[8] Aladahalli, C., Cagan, J., and Shimada, K. Objective function effect based pattern search — an implementation for 3D component layout. *Journal of Mechanical Design 129*, 3 (2007), 255–265.

[9] Aladahalli, C., Cagan, J., and Shimada, K. Objective function effect based pattern search — theoretical framework inspired by 3D component layout. *Journal of Mechanical Design 129*, 3 (2007), 243–254.

[10] Albano, A., and Sapuppo, G. Optimal allocation of two-dimensional irregular shapes using heuristic search methods. *IEEE Transactions on Systems, Man and Cybernetics SMC-10*, 5 (1980), 242–248.

[11] Alvarez-Valdes, R., Parreño, F., and Tamarit, J. A branch and bound algorithm for the strip packing problem. *OR Spectrum*, to appear.

[12] Art, Jr., R. C. An approach to the two dimensional irregular cutting stock problem. Tech. Rep. 36. Y08, IBM Cambridge Scientific Center, 1966.

[13] Aurenhammer, F. Power diagrams: Properties, algorithms and applications. *SIAM Journal on Computing 16*, 1 (1987), 78–96.

[14] Ayyadevara, V. R., Bourne, D. A., Shimada, K., and Sturges, R.H., Jr. Interference-free polyhedral configurations for stacking. *IEEE Transactions on Robotics and Automation 18*, 2 (2002), 147–165.

[15] Baker, B. S., Jr, and Rivest, R. L. Orthogonal packings in two dimensions. *SIAM Journal on Computing 9*, 4 (1980), 846–855.

[16] Bennell, J. A., and Dowsland, K. A. Hybridising tabu search with optimisation techniques for irregular stock cutting. *Management Science 47*, 8 (2001), 1160–1172.

[17] Bennell, J. A., Dowsland, K. A., and Dowsland, W. B. The irregular cutting-stock problem – a new procedure for deriving the no-fit polygon. *Computers & Operations Research 28*, 3 (2001), 271–287.

[18] Birgin, E. G., Martinez, J. M., Mascarenhas, W. F., and Ronconi, D. P. Method of sentinels for packing items within arbitrary convex regions. *Journal of the Operational Research Society 57*, 6 (2006), 735–746.

[19] Birgin, E. G., Martinez, J. M., and Ronconi, D. P. Optimizing the packing of cylinders into a rectangular container: A nonlinear approach. *European Journal of Operational Research 160*, 1 (2005), 19–33.

[20] Bischoff, E. E., and Marriott, M. D. A comparative evaluation of heuristics for container loading. *European Journal of Operational Research 44*, 2 (1990), 267–276.

[21] Blum, C., Maria, Roli, A., and Sampels, M., Eds. *Hybrid Metaheuristics: An Emerging Approach to Optimization*, 1 ed., vol. 114 of *Studies in Computational Intelligence*. Springer, 2008.

[22] Boor, V., Overmars, M. H., and van der Stappen, A. F. The gaussian sampling strategy for probabilistic roadmap planners. In *Proceedings of 1999 IEEE International Conference on Robotics and Automation* (1999), vol. 2, pp. 1018–1023.

[23] Burke, E., Hellier, R., Kendall, G., and Whitwell, G. A new bottom-left-fill heuristic algorithm for the two-dimensional irregular packing problem. *Operations Research 54*, 3 (2006), 587–601.

[24] Burke, E. K., Kendall, G., and Whitwell, G. A new placement heuristic for the orthogonal stock-cutting problem. *Operations Research 52*, 4 (2004), 655–671.

[25] Cagan, J., Shimada, K., and Yin, S. A survey of computational approaches to three-dimensional layout problems. *Computer-Aided Design 34*, 8 (2002), 597–611.

[26] Chin, F., Snoeyink, J., and Wang, C. A. Finding the medial axis of a simple polygon in linear time. *Discrete and Computational Geometry 21*, 3 (1999), 405–420.

[27] Choi, V., Agarwal, P. K., Edelsbrunner, H., and Rudolph, J. Local search heuristic for rigid protein docking. In *WABI 2004: Proceedings of the 4th Workshop on Algorithms in Bioinformatics 2004* (2004), vol. 3240 of *Lecture Notes in Computer Science*, Springer, pp. 218–229.

[28] Choi, V., and Goyal, N. A combinatorial shape matching algorithm for rigid protein docking. In *CPM2004: Proceedings of the 15th Annual Combinatorial Pattern Matching Symposium 2004* (2004), vol. 3109 of *Lecture Notes in Computer Science*, Springer, pp. 285–296.

[29] Dobkin, D., Hershberger, J., Kirkpatrick, D., and Suri, S. Computing the intersection-depth of polyhedra. *Algorithmica 9*, 6 (1993), 518–533.

[30] Dwyer, T., Marriott, K., and Stuckey, P. J. Fast node overlap removal. In *GD2005: Proceedings of the 13th International Symposium of Graph Drawing 2005* (2006), vol. 3843 of *Lecture Notes in Computer Science*, Springer, pp. 153–164.

[31] Dyckhoff, H. A typology of cutting and packing problems. *European Journal of Operational Research 44*, 2 (1990), 145–159.

[32] Egeblad, J., Nielsen, B. K., and Odgaard, A. Fast neighborhood search for two- and three-dimensional nesting problems. *European Journal of Operational Research 183*, 3 (2007), 1249–1266.

[33] Eisenbrand, F., Funke, S., Karrenbauer, A., Reichel, J., and Schömer, E. Packing a trunk: now with a twist! In *SPM 2005: Proceedings of the 2005 ACM Symposium on Solid and Physical Modeling* (2005), ACM Press, pp. 197–206.

[34] Ericson, C. *Real-Time Collision Detection*. Morgan Kaufmann, 2004.

[35] Ferrez, J.-A. *Dynamic triangulations for efficient 3D simulation of granular materials*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2001.

[36] Fletcher, R. *Practical Methods of Optimization*. Wiley, 2000.

[37] G. Di Battista, P. Eades, R. T., and Tollis, I. G. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[38] Gansner, E. R., and North, S. C. Improved force-directed layouts. In *GD 1998: Proceedings of the 6th International Symposium of Graph Drawing* (1998), vol. 1547 of *Lecture Notes in Computer Science*, Springer, pp. 364–373.

[39] Garey, M. R., and Johnson, D. S. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. W. H. Freeman, 1979.

[40] Gavrilova, M. L., and Rokne, J. O. N. Collision detection optimization in a multi-particle system. *International Journal of Computational Geometry & Applications 13*, 4 (2003), 279–301.

[41] Gensane, T., and Ryckelynck, P. Improved dense packings of congruent squares in a square. *Discrete & Computational Geometry 34* (2005), 97–109.

[42] Glover, F. W., and Kochenberger, G. A., Eds. *Handbook of Metaheuristics*, vol. 114 of *International Series in Operations Research & Management Science*. Springer, 2003.

[43] Goldberg, D., Malon, C., and Bern, M. A global approach to automatic solution of jigsaw puzzles. *Computational Geometry 28*, 2-3 (2004), 165–174.

[44] Gomes, M. A., and Oliveira, J. F. A 2-exchange heuristic for nesting problems. *European Journal of Operational Research 141*, 2 (2002), 359–370.

[45] Gomes, M. A., and Oliveira, J. F. Solving irregular strip packing problems by hybridising simulated annealing and linear programming. *European Journal of Operational Research 171*, 3 (2006), 811–829.

[46] Halperin, I., Ma, B., Wolfson, H., and Nussinov, R. Principles of docking: An overview of search algorithms and a guide to scoring functions. *Proteins 47*, 4 (2002), 409–443.

[47] Hayashi, K., Inoue, M., Masuzawa, T., and Fujiwara, H. A layout adjustment problem for disjoint rectangles preserving orthogonal order. *Systems and Computers in Japan 33*, 2 (2002), 31–42.

[48] Hifi, M. Exact algorithms for the guillotine strip cutting/packing problem. *Computers & Operations Research 25*, 11 (1998), 925–940.

[49] Hifi, M., and M'hallah, R. An exact algorithm for constrained two-dimensional two-staged cutting problems. *Operations Research 53*, 1 (2005), 140–150.

[50] Hong, S.-H., Merrick, D., and Do. The metro map layout problem. In *GD 2004: Proceedings of the 12th International Symposium on Graph Drawing*, vol. 3383 of *Lecture Notes in Computer Science*. Springer, 2005, pp. 482–491.

[51] Hopcroft, J. E., Schwartz, J. T., and Sharir, M. Efficient detection of intersections among spheres. *The International Journal of Robotics Research 2*, 4 (1983), 77–80.

[52] Hopper, E., and Turton, B. C. H. An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem. *European Journal of Operational Research 128*, 1 (2001), 34–57.

[53] Hopper, E., and Turton, B. C. H. A review of the application of meta-heuristic algorithms to 2D strip packing problems. *Artificial Intelligence Review 16*, 4 (2001), 257–300.

[54] Huang, X., and Lai, W. Force-transfer: a new approach to removing overlapping nodes in graph layout. In *ACSC 2003: Proceedings of the 26th Australasian computer science conference* (2003), Australian Computer Society, Inc., pp. 349–358.

[55] Hubbard, P. M. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics 15*, 3 (1996), 179–210.

[56] Ibaraki, T. Enumerative approaches to combinatorial optimization. *Annals of Operations Research 10-11*, 1-4 (1988), 3–342.

[57] Imahori, S., Yagiura, M., and Ibaraki, T. Local search algorithms for the rectangle packing problem with general spatial costs. *Mathematical Programming 97*, 3 (2003), 543–569.

[58] Imahori, S., Yagiura, M., and Ibaraki, T. Improved local search algorithms for the rectangle packing problem with general spatial costs. *European Journal of Operational Research 167*, 1 (2005), 48–67.

[59] Imai, H., Iri, M., and Murota, K. Voronoi diagram in the Laguerre geometry and its applications. *SIAM Journal on Computing 14*, 1 (1985), 93–105.

[60] Imamichi, T., and Nagamochi, H. A multi-sphere scheme for 2D and 3D packing problems. In *SLS 2007: Proceedings of Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics* (2007), vol. 4638 of *Lecture Notes in Computer Science*, Springer, pp. 207–211.

[61] Kavraki, L. E., Svestka, P., Latombe, J. C., and Overmars, M. H. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *Robotics and Automation, IEEE Transactions on 12*, 4 (1996), 566–580.

[62] Kenmochi, M., Imamichi, T., Nonobe, K., Yagiura, M., and Nagamochi, H. Exact algorithms for the two-dimensional strip packing problem with and without rotations. *European Journal of Operational Research*, to appear.

[63] Kim, D.-J., Guibas, L. J., and Shin, S.-Y. Fast collision detection among multiple moving spheres. *Visualization and Computer Graphics, IEEE Transactions on 4*, 3 (1998), 230–242.

[64] Kim, D.-S., Lee, B., and Sugihara, K. A sweep-line algorithm for the inclusion hierarchy among circles. *Japan Journal of Industrial and Applied Mathematics 23*, 1 (2006), 127–138.

[65] Kim, Y. J., Lin, M. C., and Manocha, D. Incremental penetration depth estimation between convex polytopes using dual-space expansion. *IEEE Transactions on Visualization and Computer Graphics 10*, 2 (2004), 152–163.

[66] Kong, W., and Kimia, B. B. On solving 2D and 3D puzzles using curve matching. In *CVPR 2001: Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2001), vol. 2, pp. 583–590.

[67] Lesh, N., Marks, J., Mcmahon, A., and Mitzenmacher, M. Exhaustive approaches to 2D rectangular perfect packings. *Information Processing Letters 90*, 1 (2004), 7–14.

[68] Leszczynski, J. S., and Ciesielski, M. Effective algorithm for detection of a collision between spherical particles. In *ICCS 2004: Proceedings of the International Conference Computational Science 2004*, vol. 3037 of *Lecture Notes in Computer Science*. Springer, 2004, pp. 348–355.

[69] Li, W., Eades, P., and Nikolov, N. Using spring algorithms to remove node overlapping. In *APVis 2005: Proceedings of the 2005 Asia-Pacific symposium on Information visualisation* (2005), Australian Computer Society, Inc., pp. 131–140.

[70] Li, Z., and Milenkovic, V. Compaction and separation algorithms for non-convex polygons and their applications. *European Journal of Operational Research 84*, 3 (1995), 539–561.

[71] Lin, M. C., and Gottschalk, S. Collision detection between geometric models: a survey. In *Proceedings of IMA Conference on Mathematics of Surfaces* (1998), pp. 37–56.

[72] Liu, D. C., and Nocedal, J. On the limited memory BFGS method for large scale optimization. *Mathematical Programming 45*, 3 (1989), 503–528.

[73] Lyons, K. A., Meijer, H., and Rappaport, D. Algorithms for cluster busting in anchored graph drawing. *Journal of Graph Algorithms and Applications 2*, 1 (1998), 1–24.

[74] Marriott, K., Stuckey, P., Tam, V., and He, W. Removing node overlapping in graph layout using constrained optimization. *Constraints 8*, 2 (2003), 143–171.

[75] Martello, S., Monaci, M., and Vigo, D. An exact approach to the strip-packing problem. *INFORMS Journal on Computing 15*, 3 (2003), 310–319.

[76] Martello, S., Pisinger, D., and Vigo, D. The three-dimensional bin packing problem. *Operations Research 48*, 2 (2000), 256–267.

[77] Milenkovic, V. J. Rotational polygon containment and minimum enclosure using only robust 2D constructions. *Computational Geometry 13*, 1 (1999), 3–19.

[78] Misue, K., Eades, P., Lai, W., and Sugiyama, K. Layout adjustment and the mental map. *Journal of Visual Languages & Computing 6*, 2 (1995), 183–210.

[79] Nielsen, B. K., and Odgaard, A. Fast neighborhood search for the nesting problem. Tech. Rep. 03/02, DIKU, Department of Computer Science, University of Copenhagen, 2003.

[80] Nocedal, J., and Wright, S. J. *Numerical Optimization*, second ed. Springer Series in Operations Research and Financial Engineering. Springer, 2006.

[81] Oliveira, J. F., Gomes, A. M., and Ferreira, J. S. TOPOS – a new constructive algorithm for nesting problems. *OR Spektrum 22*, 2 (2000), 263–284.

[82] Overmars, M. H. Point location in fat subdivisions. *Information Processing Letters 44*, 5 (1992), 261–265.

[83] Palmer, I. J., and Grimsdale, R. L. Collision detection for animation using sphere-trees. *Computer Graphics Forum 14*, 2 (1995), 105–116.

[84] Pisinger, D. Heuristics for the container loading problem. *European Journal of Operational Research 141*, 2 (2002), 382–392.

[85] Preparata, F. P., and Shamos, M. I. *Computational Geometry: An Introduction*. Springer-Verlag New York, Inc., 1985.

[86] Ramkumar, G. D. An algorithm to compute the minkowski sum outer-face of two simple polygons. In *SCG 1996: Proceedings of the 12th Annual ACM Symposium on Computational Geometry* (1996), ACM Press, pp. 234–241.

[87] Shimada, K., and Gossard, D. C. Bubble mesh: automated triangular meshing of non-manifold geometry by sphere packing. In *SMA '95: Proceedings of the 3rd ACM Symposium on Solid Modeling and Applications* (1995), ACM Press, pp. 409–419.

[88] Sloane, N. J. A. The sphere packing problem. *Documenta Mathematica ICM III* (1998), 387–396.

[89] Taylor, R. D., Jewsbury, P. J., and Essex, J. W. A review of protein-small molecule docking methods. *Journal of Computer-Aided Molecular Design 16*, 3 (2002), 151–166.

[90] Turek, J., Wolf, J. L., and Yu, P. S. Approximate algorithms for scheduling parallelizable tasks. In *SPAA 1992: Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures* (1992), ACM Press, pp. 323–332.

[91] Turk, G. Interactive collision detection for molecular graphics. Tech. rep., University of North Carolina at Chapel Hill, 1990.

[92] Vemuri, B. C., Chen, L., Vu-Quoc, L., Zhang, X., and Walton, O. Efficient and accurate collision detection for granular flow simulation. *Graphical Models and Image Processing 60*, 6 (1998), 403–422.

[93] Wang, H., Huang, W., Zhang, Q., and Xu, D. An improved algorithm for the packing of unequal circles within a larger containing circle. *European Journal of Operational Research 141*, 2 (2002), 440–453.

[94] Wäscher, G., Haussner, H., and Schumann, H. An improved typology of cutting and packing problems. *European Journal of Operational Research 183*, 3 (2007), 1109–1130.

[95] Winterfeld, A. Application of general semi-infinite programming to lapidary cutting problems. Tech. Rep. 91, Fraunhofer-Instituts für Techno- und Wirtschaftsmathematik, 2006.

[96] Wolfson, H., Schonberg, E., Kalvin, A., and Lamdan, Y. Solving jigsaw puzzles by computer. *Annals of Operations Research 12*, 1 (1988), 51–64.

[97] Yuping, Z., Shouwei, J., and Chunli, Z. A very fast simulated re-annealing algorithm for the leather nesting problem. *The International Journal of Advanced Manufacturing Technology 25*, 11 (2005), 1113–1118.

List of Publications

Journals

1. Imamichi T., and Nagamochi H. Performance analysis of a collision detection algorithm of spheres based on slab partitioning, *The IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Special Section on Discrete Mathematics and Its Applications,* vol. E91-A, no. 9, pp. 2308–2313, 2008.

International Conferences with Review

1. Imamichi T., Arahori Y., Gim J., Hong S.-H., and Nagamochi H. Removing node overlaps using multi-sphere scheme, In *Proceedings of the 16th International Symposium on Graph Drawing (GD2008),* September 21–24, 2008, Heraklion, Crete, Greece, Lecture Notes in Computer Science, Springer, vol. 5417 pp. 296–301, to appear.

2. Imamichi T., and Nagamochi H. A multi-sphere scheme for 2D and 3D packing problems, In *Proceedings of Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics (SLS2007)*, September 6–8, 2007, Universite Libre de Bruxelles, Brussels, Belgium, Lecture Notes in Computer Science, Springer, vol. 4638, pp. 207–211.

3. Imamichi T., Yagiura M., and Nagamochi H. An iterated local search algorithm based on nonlinear programming for the irregular strip packing problem, In *Proceedings of International Symposium on Scheduling 2006 (ISS2006)*, Tokyo, Japan, July 18–20, 2006, pp. 132–137.

Unpublished Manuscripts

1. Imamichi T., Yagiura M., and Nagamochi H. An iterated local search algorithm based on nonlinear programming for the irregular strip packing problem, submitted for publication.